Proyecto Fin de Carrera
PLAN 2000

E.U.I.T. TELECOMUNICACIÓN

TEMA: Codificación de video

TÍTULO: Caracterización de un descodificador HEVC ejecutándose en un DSP

AUTOR: Jesús Pablo Caño Velasco

TUTOR: Fernando Pescador del Oso **VºBº.**

DEPARTAMENTO: SEC

Miembros del Tribunal Calificador:

PRESIDENTE:

VOCAL:

VOCAL SECRETARIO:

DIRECTOR:

Fecha de lectura:

Calificación: **El Secretario,**

RESUMEN DEL PROYECTO:

HEVC es el nuevo estándar de codificación de vídeo que está siendo desarrollado conjuntamente por las organizaciones *ITU-T Video Coding Experts Group* (VCEG) e *ISO/IEC Moving Picture Experts Group* (MPEG). Su objetivo principal es mejorar la compresión de vídeo, en relación a los actuales estándares.

Es común hoy en día, debido a su flexibilidad para aplicaciones de bajo consumo, diseñar sistemas de descodificación de vídeo basados en un procesador digital de señal (DSP). En la mayoría de las veces, los diseños parten de un código creado para ser ejecutado en un ordenador personal y posteriormente se optimizan para tecnología DSP.

El objetivo principal de este proyecto es caracterizar el rendimiento de un sistema basado en DSP que ejecute el código de un descodificador de video HEVC.

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

Caracterización de un descodificador HEVC ejecutándose en un DSP

Autor:
Jesús Pablo CAÑO VELASCO
jp.cano@alumnos.upm.es

Tutor:
Fernando PESCADOR DEL OSO
pescador@sec.upm.es

Resumen

HEVC es el nuevo estándar de codificación de vídeo que está siendo desarrollado conjuntamente por las organizaciones *ITU-T Video Coding Experts Group* (VCEG) e *ISO/IEC Moving Picture Experts Group* (MPEG). Su objetivo principal es mejorar la compresión de vídeo, en relación a los actuales estándares.

Es común hoy en día, debido a su flexibilidad para aplicaciones de bajo consumo, diseñar sistemas de descodificación de vídeo basados en un procesador digital de señal (DSP). En la mayoría de las veces, los diseños parten de un código creado para ser ejecutado en un ordenador personal y posteriormente se optimizan para tecnología DSP.

El objetivo principal de este proyecto es caracterizar el rendimiento de un sistema basado en DSP que ejecute el código de un descodificador de video HEVC.

Keywords: HEVC, rendimiento, DSP, codificación de video.

Abstract

HEVC is a new video coding standard which is being developed by both *ITU-T Video Coding Experts Group* (VCEG) and *ISO/IEC Moving Picture Experts Group* (MPEG). Its main goal is to improve video compression, compared with the actual standards.

It is common practice, because of the flexibility in low power applications, to design video decoding systems using digital signal processors (DSP). Most of the time, these designs start with a code suitable to be executed in personal computers and then it is optimized for DSP technology.

The main goal in this final degree project is to characterize the performance of a DSP based system executing an HEVC video decoder.

Keywords: HEVC, performance, DSP, video coding.

Agradecimientos

Índice general

Índice de figuras

Índice de tablas

Índice de listados

Capítulo 1

Introducción

Desde 1984 se han desarrollado técnicas de codificación que tienen como objetivo reducir la redundancia de los datos en video digital. Éstas técnicas se han materializado en diversos estándares desarrollados por los organismos de estandarización ITU-T VCEG y ISO/IEC MPEG (ver figura 1.1). Todas éstas técnicas se basan en un esquema de codificación híbrido cuya finalidad es reducir la redundancia espacial y temporal existente en las secuencias de video.

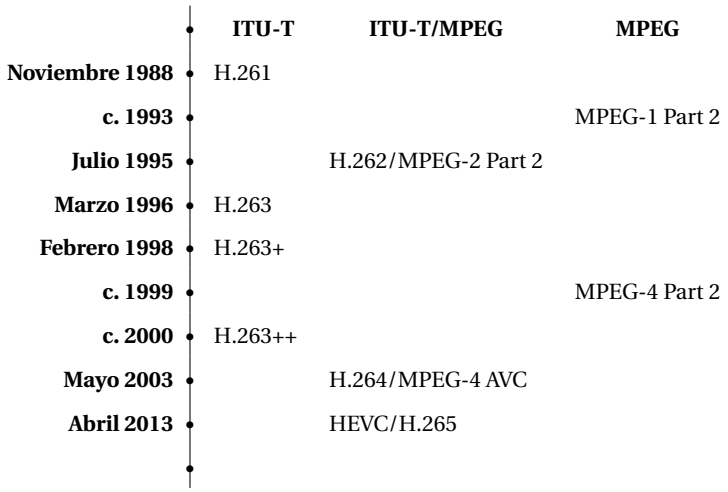


Figura 1.1: Cronología de los principales estándares de codificación de vídeo.

Actualmente HEVC (High Efficiency Video Coding) es el proyecto más reciente de JCT-VC[1] (Join Collaborative Team on Video Coding). Se trata del nuevo estándar de codificación de video que permite reducir hasta un 50 % el regímen binario necesario para la codificación de secuencias en alta calidad si lo comparamos con el actual estándar H.264. Puede soportar resoluciones de hasta 8K (8192x4320) UHD (*Ultra High Definition*) y para su diseño se ha pensado en los sistemas con arquitectura multinúcleo.

En Abril de 2013, HEVC fue aprobado como estándar por la ITU-T con el nombre de H.265. En Junio de este mismo año el estándar fue publicado formalmente en la web de ITU-T [10]. Se espera que ISO/IEC apruebe HEVC con el nombre de MPEG-H Part 2 (ISO/IEC 23008-2).

Paralelamente al desarrollo de estos estándares se inició el proyecto *openHEVC* [3]. Se trata de un descodificador HEVC optimizado y de software libre que mejora en rendimiento al descodificador de referencia HM [8].

Un software para la descodificación de vídeo requiere ejecutarse en soportes tecnológicos con una capacidad de cómputo elevada. Y es aún más necesario para la codificación de vídeo. En la tabla 1.1 se muestra una comparación básica de las principales alternativas tecnológicas que se pueden usar actualmente para implementar estas aplicaciones: las basadas en procesadores de propósito general (GPP), las basadas en procesadores digital de señal (DSP) y las basadas en arquitecturas específicas.

[1] JCT-VC es el organismo resultado de los trabajos conjuntos de ITU-T VCEG e ISO/IEC MPEG.

Tabla 1.1: Comparación de las plataformas para la descodificación de video.

Característica	GPP	DSP	Arq. Específicas
Consumo	Alto	Medio	Bajo
Rendimiento	Bajo	Medio	Alto
Costes fijos	Medio	Bajo	Alto
Costes variables	Alto	Alto	Bajo
Programabilidad	Alto	Alto	Bajo
Tiempo desarrollo	Bajo	Bajo	Alto

Las plataformas con procesadores digital de señal son inferiores a las de arquitecturas especificas si tenemos en cuenta su mayor consumo y costes a gran escala. Sin embargo su arquitectura programable permite un desarrollo rápido de aplicaciones y por lo tanto son más flexibles.

Debido a la versatilidad de las plataformas con procesadores digital de señal éstas presentan un gran atractivo para el desarrollo de determinadas aplicaciones. Por ello, el GDEM[2] comenzó a investigar en 2004 formas de implementar codificadores y descodificadores de vídeo en estas plataformas. El presente proyecto fin de carrera se enmarca dentro de esta línea de trabajo.

1.1. Objetivos

Este proyecto fin de carrera está motivado por el objetivo de caracterizar el rendimiento de un sistema basado en DSP en la descodificación de video HEVC.

Para llevarlo a cabo, en este proyecto se han realizado las siguientes tareas:

- Migrar el descodificador openHEVC del PC a la tarjeta de prototipado basada en DSP DM6437 EVM.
- Definir un banco de pruebas automatizado para obtener medidas del rendimiento del sistema DSP.
- Procesar los datos obtenidos y sacar conclusiones.
- Comparar el rendimiento del openHEVC con otros descodificadores HEVC como HM9.0.

1.2. Alcance

Este proyecto fin de carrera tiene como fin evaluar el rendimiento de openHEVC sobre la tarjeta de prototipado basada en DSP DM6437 EVM. Se han realizado las modificaciones mínimas necesarias en el código de openHEVC para portarlo al DSP. Sin embargo no se han realizado las optimizaciones necesarias para aprovechar las características del hardware del DSP.

1.3. Organización de la memoria

El capítulo 2 resume las principales características del estándar de codificación de vídeo HEVC incluyendo las diferencias más sobresalientes con respecto al anterior estándar H.264.

[2] Grupo de Diseño Electrónico y Microelectrónico de la EUITT-UPM.

El capítulo 3 presenta una descripción de la plataforma *hardware* que se ha utilizado en este proyecto así como una descripción de las principales plataformas tecnológicas sobre las que se pueden implementar descodificadores de video en la actualidad.

El capítulo 4 cubre con detalle todo el proceso que se ha llevado a cabo para medir el rendimiento del descodificador openHEVC en la plataforma DSP incluyendo los resultados obtenidos y una comparación con el rendimiento del descodificador HM 9.0 obtenido en otro trabajo.

El capítulo 5 presenta las conclusiones obtenidas de los resultados del capítulo 4. Por último, se exponen las limitaciones de este proyecto fin de carrera y se presentan como líneas de trabajo futuro.

Capítulo 2

HEVC

En este capítulo se resumen las características más sobresalientes del estándar de codificación de vídeo HEVC. La exposición se centra en el proceso de descodificación y en las diferencias que este nuevo estándar presenta con respecto al anterior H.264/MPEG-4 AVC.

En ningún caso se pretende realizar un estudio exhaustivo puesto que ya existen diversas publicaciones [13, 21] donde se describe HEVC con más detalle. En [24] se puede encontrar un estudio aún más detallado de este tema. Además, el texto de la recomendación de ITU-T se encuentra disponible públicamente en [10].

En la sección 2.1 se presenta una introducción a HEVC. Un descripción del funcionamiento de un descodificador HEVC se resume en la sección 2.2. La sección 2.3 trata del particionameinto de las imágenes. Por último, en las secciones 2.4, 2.5, 2.6, 2.7, 2.8 y 2.9 se tratan algunas de las principales herramientas de las que se compone HEVC.

2.1. Introducción

Debido a los avances tecnológicos de la última década, se ha incrementado la popularidad del vídeo HD (*High Definition*) y UHD (*Ultra High Definition*). Actualmente Internet y otras redes de comunicación no tienen la suficiente capacidad para transmitir grandes cantidades de contenido alta definición. Por este motivo, es necesario desarrollar nuevos sistemas de codificación de vídeo que permitan reducir el régimen binario necesario para la transmisión de este contenido, manteniendo la misma calidad visual.

HEVC (*High Efficency Video Coding*) es el nuevo estándar de compresión de video que está siendo desarrollado conjuntamente por ITU-T VCEG (*Video Coding Expertes Group* e ISO/IEC MPEG (*Moving Pictures Expert Group*) a través de la organización conjunta JCT-VC (*Joint Colaborative Team on Video Coding*).

Según algunos análisis comparativos realizados durante el proceso de estandarización, HEVC permite reducir en un 50 % el régimen binario en secuencias HD (1280x720), manteniendo la misma calidad visual que se pude obtener con el estándar actual, H.264/MPEG-4 AVC (*Advanced Video Coding*). Sin embargo se espera que el codificador HEVC sea varios órdenes de magnitud más complejo [5, 33].

Con respecto al descodificador, algunos estudios calculan que es entre 1.5 y 2 veces más complejo en relación al descodificador H.264 [33, 34], mientras que otros análisis concluyen que la complejidad de los descodificadores HEVC no es tan diferente [5].

Algunas de las características que diferencian a HECV de otros estándares de compresión de vídeo son:

- Particionamiento *quadtree* de las imágenes, con tamaños de bloques desde 64x64 hasta 8x8 *pixels*.
- Soporte para codificación y descodificación en paralelo, usando *tiles* y WPP (*Wavefront Paralell Processing*).
- Más modos de predicción *intra* (35 en total, 33 de ellos direccionales).

- Soporte para transformadas en bloques desde 32x32 hasta 8x8 pixels, además de bloques rectangulares.
- Bloque de compensación de movimiento mejorado, incluyendo un nuevo modo *merge*.
- Inclusión del nuevo filtro SAO (*Sampling Adaptive Offset*) en el lazo.

2.2. Diseño de un descodificador HEVC

HEVC está basado en el mismo esquema de codificación "híbrido" (predicción *inter-intra* y transformada bidimensional) usado en todos los estándares de compresión de vídeo desde H.261. El nuevo estándar no es un diseño revolucionario, más bien se trata de una serie de pequeñas mejoras realizadas a H.264 para conseguir reducir el régimen binario.

En la figura 2.1 se muestra el diagrama de bloques simplificado de un descodificador HEVC. El descodificador de entropía identifica cada una unidad NAL (*Network Abstraction Layer*) y descodifica el residuo, los vectores de movimiento, el modo de predicción u otra información contextual contenida en ella. Posteriormente los coeficientes del residuo se transforman del dominio de las frecuencias espaciales al dominio espacial por medio de una transformada basada en la DCT (*Discrete Cosine Transform*). Los coeficientes transformados se suman a una predicción espacial o temporal, según se trate del modo *intra* o *inter* respectivamente. Posteriormente a la imagen obtenida se le aplican los filtros antibloques y SAO con el fin de eliminar ciertas distorsiones inherentes al proceso de codificación. Finalmente la imagen obtenida se guarda en la memoria de imágenes descodificadas.

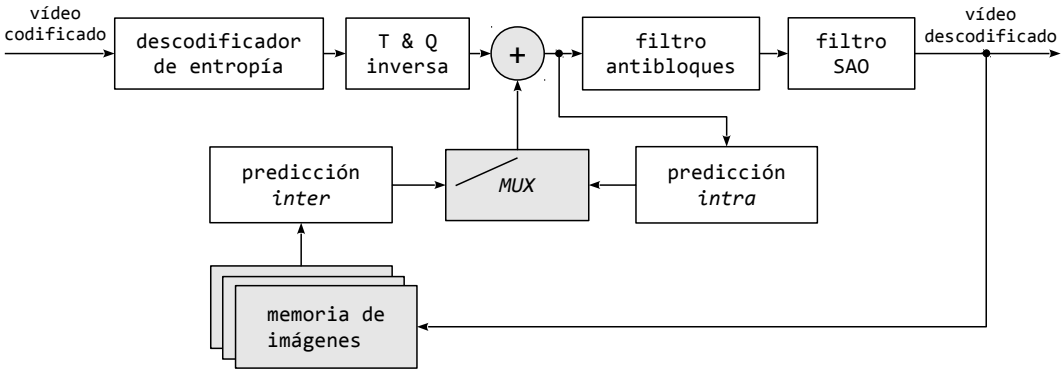


Figura 2.1: Diagrama de bloques simplificado de un descodificador HEVC.

2.3. Particionamiento de la imagen

Como ocurre en otros estándares, HEVC tiene un esquema de codificación basado en bloques. Como se verá en las siguientes subsecciones, la flexibilidad que ofrece HEVC para el particionamiento es una de las características que más contribuye a la compresión de las imágenes en este nuevo estándar. A continuación se describen los diferentes tipos de bloques y la jerarquía entre ellos.

2.3.1. Coding Units

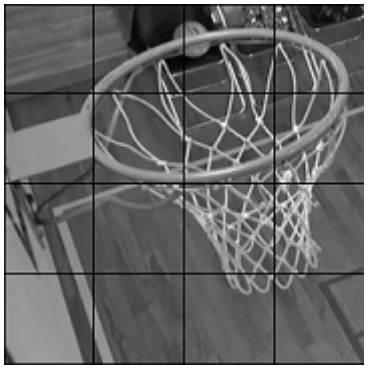
En HEVC, una imagen se divide en diferentes CTUs (*Coding Tree Units*) que pueden tener un tamaño máximo de LxL píxeles, con L igual a 16, 32 ó 64 para el caso de la luminancia (ver figura

2.2a). Cada CTU contiene tres CTBs (*Coding Tree Blocks*), uno representa la información de la luminancia y los otros dos las de las crominancias.

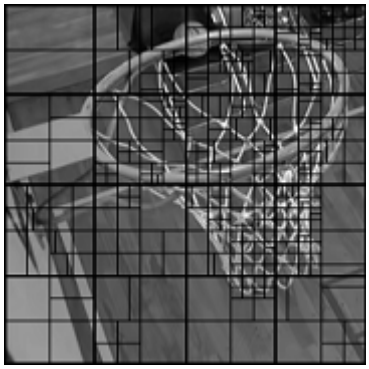
Comparación con H.264

La noción de CTU en HEVC es similar a la de los macrobloques en H.264 aunque en éste último los macrobloques tienen un tamaño de 16x16 píxeles. Generalmente los bloques grandes se comportan mejor en las zonas planas de la imágenes mientras que los bloques pequeños son útiles en las zonas con más detalles. Con el advenimiento de vídeos con resoluciones HD y UHD, la probabilidad de que una imagen contenga grandes zonas planas es mayor y por ello HEVC utiliza bloques con tamaños mayores.

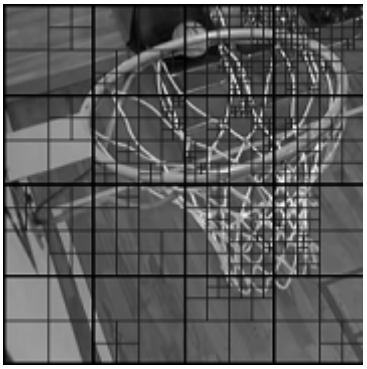
Por otra parte, cada CTB se puede dividir en CBs (*Coding Blocks*) con un tamaño máximo igual al de la CTU y un tamaño mínimo de 8x8 píxeles. Un CB de luminancia y los dos CBs de crominancias junto con la información sintáctica asociada forman una CU (*Coding Unit*). La división de las CTUs en CUs (*Coding Unit*) se realiza con una estructura de datos jerárquica llamada *quadtree*[1].



(a) Imagen dividida en CTUs de 64x64 pixeles.



(b) Imagen dividida en PBs de diferente tamaño. Se permiten bloques rectangulares (ej. 62x32 o 16x4).



(c) Imagen dividida en TBs luma con un tamaño de 32x32 a 4x4 pixeles.

Figura 2.2: Division de una imagen en diferentes tipos de bloques.

[1] Es un tipo de árbol que sirve para describir clases de estructuras de datos jerárquicas cuya propiedad común es que están basados en el principio de descomposición recursiva del espacio.

2.3.2. Prediction Units

Cada CU se pude dividir en otros bloques llamados PUs (*Predicion Unit*). En este caso la división de las CUs no es recursiva, de forma que sólo se puede realizar una vez. La función de las PUs es indicar el modo de predicción (intra o *inter*) de la zona que representan.

Las PUs pueden tener un tamaño desde 64x64 hasta 4x4 píxeles y como se observa en la figura 2.3 pueden ser simétricas o asimétricas. Las PUs del tipo M/2xM/2 sólo se permiten cuando la CU tiene el tamaño mínimo (8x8 píxeles). Por otra parte, en el caso de que la predicción sea *intra*, sólo se permiten PUs simétricas. En la figura 2.2b se muestra un ejemplo de la división de una imagen en diferentes PUs.

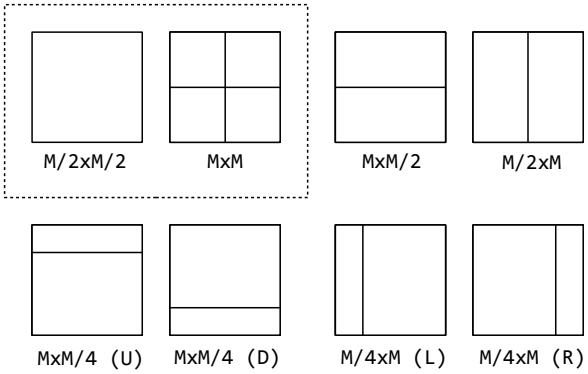


Figura 2.3: Modos de particionamiento de PUs. A las CUs *inter* se le pueden aplicar cualquiera de los modos, mientras que a las CUs *intra* sólo se le pueden aplicar las dos primeras (M/2xM/2 ó MxM).

2.3.3. Transform Units

Como en otros estándares, HEVC aplica una transformada del tipo DCT a los residuos de las imágenes. La unidad básica que se utiliza para este proceso es la TU (*Transform Unit*). La TU pude tener un tamaño igual a la CU asociada o bien dividirla recursivamente, siempre de forma simétrica. El tamaño máximo que puede tener es 32x32 y el mínimo 4x4 píxeles. En la figura 2.2c se muestra la división de una imagen es sus diferentes TUs.

Comparación con H.264

A diferencia de H.264, en HEVC un TB pude contener múltiples PBs cuando el modo de predicción es inter. *Este hecho se puede comprobar si se observan detenidamente las figuras 2.2b y 2.2c.*

2.3.4. Slices

En HEVC, una imagen se puede dividir en diferentes *slices* que pueden ser descodificadas independientemente de otras *slices* de la misma imagen en términos de entropía, predicción y reconstrucción de los residuos. Una *slice* puede ser una región de una imagen, como se muestra en la figura 2.4, o bien puede ser la imagen completa.

Uno de las principales funciones que desempeñan las *slices* es la resincronización de la descodificación del vídeo en caso de una pérdida de datos.

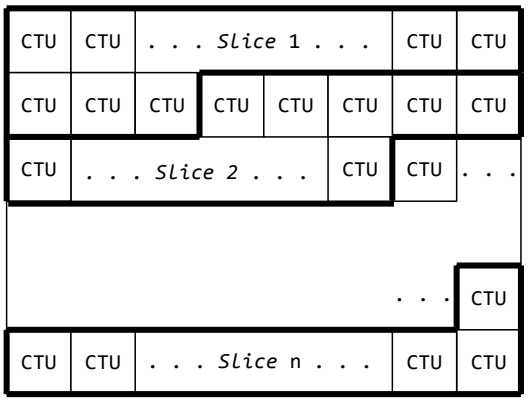


Figura 2.4: Subdivisión de una imagen en *slices*.

Comparación con H.264

Las slices *en HEVC son similares a las del estándar H.264 con la diferencia de que no disponen de FMO (*Flexible Macroblock Ordering*). FMO es una herramienta de H.264 para situar los macrobloques en la* slices *de una manera flexible. Sin embargo, FMO no se ha usado de forma exitosa debido a la complejidad que añade y por la disminución de la eficiencia en la codificación al deshabilitar la predicción entre los bordes del* slice.

2.3.5. Tiles

En la figura 2.5 se muestra una imagen dividida en *tiles*. Las *tiles* son regiones rectangulares de la imagen, típicamente con el mismo número de CTUs, que pueden ser descodificadas independientemente y que comparten algunas cabeceras de información.

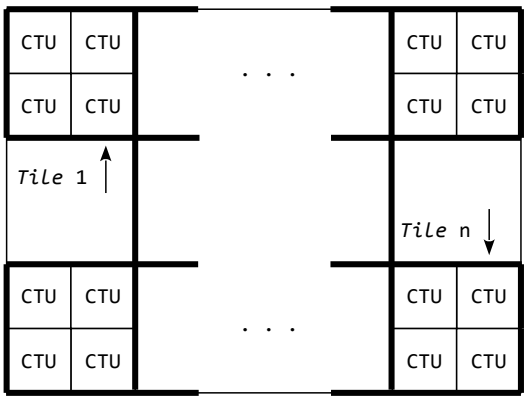


Figura 2.5: Subdivisión de una imagen en *tiles*.

La principal función de las *tiles* es la de proporcionar cierto nivel de paralelización en los procesos de codificación y descodificación de vídeo que puede ser explotado en sistemas con arquitecturas de procesamiento en paralelo (p. ej. en DSPs multinúcleo).

Comparación con H.264

Las tiles *son más flexibles que las* slices *de H.264 para el procesamiento en paralelo y es una tecnología considerablemente menos compleja que FMO.*

2.3.6. WPP

Cuando WPP (*Wavefront Parallel Processing*) está habilitado, una *slice* se divide en filas de CTUs. Así la primera fila puede ser procesada por un hilo de ejecución; un segundo hilo de ejecución puede comenzar con la segunda fila tras haberse procesado las dos primeras CTUs de la fila anterior. Este mismo proceso continúa para la siguiente fila tal y como indica la figura 2.6.

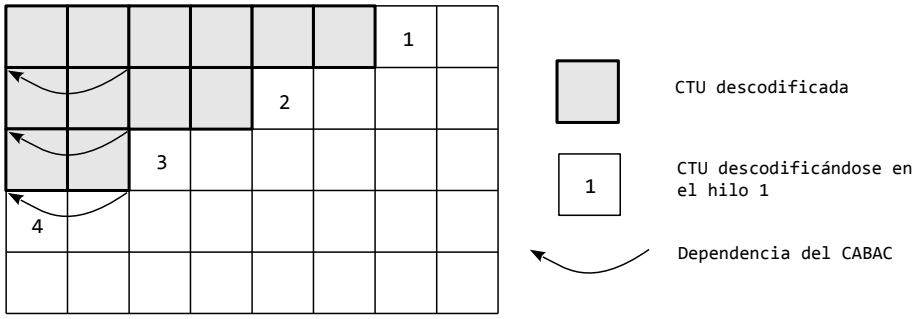


Figura 2.6: Procesamiento de una imagen con WPP habilitado.

Las dos CTUs de la fila anterior son requeridas puesto que se depende de su información para la predicción *intra* y para el cálculo de los vectores de movimiento en la fila actual. Además, los parámetros de la entropía se inicializan en base a la información obtenida en la fila anterior.

Al igual que las *tiles*, las WPPs proporcionan una forma de paralelización en los procesos de codificación y descodificación. La diferencia entre ambas es que con las WPPs se consigue un mejor rendimiento y además no se introducen determinadas aberraciones visuales como puede ocurrir en las *tiles*.

2.4. Predicción *intra*

HEVC usa la predicción *intra* para aprovechar la redundancia espacial existente en cualquier secuencia y, con ello, reducir el régimen binario necesario para su codificación. Para ello se usan muestras de TBs adyacentes como datos de referencia para la predicción. Este proceso se realiza a nivel de PU y la forma de hacerlo varía según sea el modo de predicción asociado a dicha PU. Los modos son: *Intra_Angular*, *Intra_Planar* e *Intra_DC*.

En los modos *Intra_Angular*, cada TB se predice sobre muestras adyacentes en una dirección determinada. A modo informativo, en la figura 2.7 se muestran las 33 direcciones diferentes que se pueden usar. Los ángulos se han diseñado intecionadamente para conseguir cubrir con mayor densidad las zonas próximas a la dirección horizontal, vertical y diagonal puesto que se ha observado estadísticamente la prevalencia de estos ángulos.

En el modo *Intra_DC* la predicción se genera con el valor medio de las muestras de referencia mientras que en el modo *Intra_Planar* se usa el valor medio de dos interpolaciones lineales (vertical y horizontal). El modo *Intra_Planar* es útil en la predicción de zonas planas de la imagen.

Para la predicción de las crominancias, HEVC permite al codificador seleccionar cinco modos diferentes: vertical, horizontal, *Intra_Planar*, *Intra_DC* y el mismo modo usado en la luminancia.

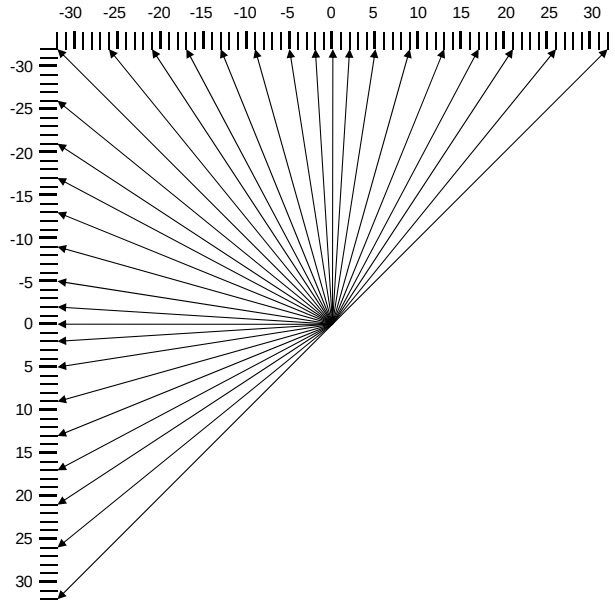


Figura 2.7: Muestra de los 33 ángulos definidos en la predicción *intra*.

Debido a la existencia de un gran número de modos, éstos se codifican de forma predictiva considerando los tres modos más probables (MPM, *Most Probable Modes*) basados en los PBs adyacentes descodificados previamente.

Comparación con H.264

HEVC dispone de 33 modos de predicción direccionales para la luminancia en contraste con los 8 de H.264. Además el proceso de predicción en el modo Intra_Angular *es consistente para cualquier tamaño de bloque mientras que en H.264 se usan diferentes métodos según el tamaño de éstos. Todo ello hace que en HEVC la predicción* intra *sea más flexible que en los anteriores estándares.*

2.5. Predicción *inter*

Las imágenes *inter* son aquellas codificadas con referencia a otras imágenes. La predicción *inter* explota las similitudes de cada imagen con sus vecinas en el dominio temporal.

Cada PU codificada en modo *inter* puede tener dimensiones cuadradas o rectangulares (como se mostró en la figura 2.3). Las dimensiones rectangulares son útiles porque permiten a las PUs adaptarse con más precisión a la forma de los objetos sin que sea necesario dividirlas en otras PUs más pequeñas.

En la compensación de movimiento se utiliza una precisión de 1/4 de píxel usando filtros 7-tap u 8-tap. Además pueden existir hasta 6 imágenes de referencia o , si se trabaja con resoluciones bajas, se pueden usar hasta 16.

Comparación con H.264

En H.264 se utilizan filtros 6-tap para las posiciones de 1/2 de píxel mientras que las posiciones 1/4 se calculan aplicando una interpolación bilineal sobre los valores de 1/2 píxel y píxeles enteros. En HEVC no se realizan estas interpolaciones ya que se utilizan filtros de mayor orden.

Por cada PU se pueden transmitir uno o dos vectores de movimiento, según se trate de predicción unidireccional o bidireccional respectivamente. Además, con el fin de mejorar la eficiencia en la codificación, se puede utilizar la predicción de vectores de movimiento basándose en los candidatos más probables utilizando los datos de los PBs adyacentes en las imágenes de referencia.

2.6. Transformada y cuantificación

Como en otros estándares, HEVC aplica una transformada del tipo DCT (*Discrete Cosine Transforme*) durante la codificación de los bloques TB. A continuación se muestra una de las posibles transformadas DCT que se pueden utilizar, particularizada para bloques de 32x32 píxeles y normalizada.

$$f(u,v) = \frac{1}{16}C(u)C(v)\sum_{x=0}^{31}\sum_{y=0}^{31}F(x,y)cos\left[\frac{\pi u(2x+1)}{64}\right]cos\left[\frac{\pi v\left(2y+1\right)}{64}\right] \tag{2.1}$$

donde x e y son las coordenadas en el dominio espacial; u y v son las coordenadas en el domino de las frecuencias espaciales; por último $C(u)$ y $C(v)$ se definen de la siguiente forma:

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } u = 0 \\ 1 & \text{si } u \neq 0 \end{cases} \qquad C(v) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } v = 0 \\ 1 & \text{si } v \neq 0 \end{cases} \tag{2.2}$$

Concretamente la matriz de transformación 32x32 que se ha diseñado en HEVC (*core transform*) resulta de una aproximación de la ecuación 2.1. Las matrices de tamaño 16x16, 8x8 y 4x4 se pueden obtener a partir de la de 32x32.

Alternativamente, para TUs de tamaño 4x4 píxeles y predicción *intra*, se utiliza una transformada derivada de una aproximación de la DST (*Discret Sine Transform*) ya que con ello se logra reducir un 1 % el régimen binario.

2.7. Codificador de entropía

Después de la transformada y la cuantificación, se codifican todos los elementos sintácticos y los coeficientes ya transformados. Para ello se usa un codificador de entropía que aprovecha las propiedades estadísticas de los datos y asigna códigos de menor longitud a aquellos elementos que son más probables, reduciendo de este modo el régimen binario.

La mayoría de los elementos se codifican usando CABAC (*Context Adaptative Binary Arithmetic Coding*). Tan sólo algunos elementos, como por ejemplo las cabeceras de las *slices* y algunas unidades NAL, se codifican mediante códigos de longitud variable *Exp-Golomb*.

Comparación con H.264

*En H.264 el principal codificador de entropía es del tipo CAVLC (*Context Adaptative Variable Lenght Coding*) mientras que CABAC se usa opcionalmente en los perfiles* main *y* high. *En HEVC, el codificador CABAC se ha mejorado con respecto al usado en H.264 mediante la reducción del número de contextos y la adicción de mejoras para el procesamiento en paralelo.*

2.8. Filtro antibloques

Como en cualquier otro estándar de codificación de vídeo basado en bloques, con el fin de mejorar la calidad subjetiva y objetiva del vídeo, en HEVC se aplica un filtro paso-bajo en los contornos de los bloques una vez que éstos se han descodificado.

Comparación con H.264

En HEVC el diseño del filtro antibloques se ha simplificado ya que las operaciones de filtrado no se realiza en los bloques de tamaño 4x4. Además, este filtro se ha adaptado para ser usado en sistemas con capacidad para el procesamiento en paralelo.

2.9. Filtro SAO

Una de las razones por las que HEVC mejora la eficiencia en la codificación con respecto a estándares anteriores es el mayor tamaño de los bloques TB (hasta 32x32 píxeles). Sin embargo esto tiende a causar distorsiones no deseadas conocidas como *ringing* y *banding*.

El filtro SAO (*Sampling Adaptative Offset*) es una nueva herramienta que se ha añadido a HEVC para mejorar el aspecto visual de la imagen suprimiendo estas distorsiones. En la figura 2.8 se muestra un ejemplo de cómo el filtro SAO ayuda a disminuir los efectos que se producen en los bordes de los objetos.

(a) SAO no habilitado. **(b)** SAO habilitado.

Figura 2.8: Ejemplo de mejora visual con el filtro SAO. [7]

Cuando el filtro SAO está habilitado se clasifican los píxeles de una región en diferentes categorías basándose en su intensidad o en sus propiedades de contorno. Después, con el fin de reducir la distorsión, se añade un valor de *offset*, *band offset* (BO) o *edge offset* (EO), a cada uno de esos píxeles según la categoría en la que se encuentran.

Capítulo 3

La plataforma de desarrollo

En este capítulo se describe la plataforma sobre la que se ha implementado el descodificador de video HEVC. En la sección 3.1 se muestran algunas de las principales plataformas sobre las que se pueden ejecutar descodificadores de vídeo en la actualidad. En las secciones 3.2 y 3.3 se habla sobre la plataforma *hardware* que se ha usado en este proyecto. Por último, en la sección 3.4 se exponen algunas de las características de *Code Composser Studio*, el entorno de desarrollo integrado que se ha usado.

3.1. Alternativas tecnológicas

En los últimos años se han venido utilizando diferentes plataformas para implementar aplicaciones de descodificación de video. En [19] se analizan con detalle estas soluciones. A continuación se resumen algunas de las propuestas más representativas:

- **Alternativas basadas en procesadores de propósito general.** En este grupo se suelen encontrar las primeras implementaciones de los diversos estándares de codificación para plataformas basadas en ordenador personal [8]. Estas implementaciones, generalmente programadas en C o C++, no están optimizadas para ninguna arquitectura concreta y suelen ser el punto de partida para futuros descodificadores. La utilización de un procesador de propósito general no es adecuado para plataformas de bajo coste o portátiles puesto que es difícil encontrar un compromiso adecuado entre potencia de cálculo y coste y consumo reducidos. En este apartado cabe destacar las implementaciones basadas en procesadores Intel, ARM y CELL.

- **Alternativas basadas en arquitecturas específicas.** Dentro de este grupo se incluyen un conjunto de implementaciones heterogéneas que emplean arquitecturas especialmente diseñadas para minimizar el tiempo de descodificación y el consumo. Estas propuestas poseen una elevada eficiencia aunque son poco flexibles puesto que una vez fabricadas son difíciles de modificar. Algunos ejemplos son implementaciones en FPGAs y ASICs.

- **Alternativas basadas en procesadores digitales de señal.** Los DSP (Procesadores Digitales de Señal) tienen una arquitectura diseñada específicamente para ser utilizados en sistemas multimedia. Además los periféricos que incluyen son muy útiles en sistemas de codificación o descodificación de video. Esta propuesta se caracteriza por aunar la flexibilidad de los procesadores de propósito general y las prestaciones de las arquitecturas específicas manteniendo un coste y consumo energético bajo.

En este proyecto se ha optado por la última propuesta, utilizándose el DSP TMS320DM6437 [28] de Texas Instruments como soporte para implementar el descodificador openHEVC [3]. Su elección se basa en las prestaciones que posee, las herramientas de desarrollo disponibles y la experiencia acumulada en el grupo de investigación GDEM[1] en el que se ha realizado este proyecto fin de carrera.

[1] Grupo de Diseño Electrónico y Microelectrónico del Departamento de Sistemas Electrónicos y de Control de la EUITT.

3.2. La tarjeta DM6437EVM

La tarjeta DM6437 [22, 23](ver figura 3.1) es una plataforma de desarrollo, fabricada por Spectrum Digital, que permite desarrollar y evaluar aplicaciones para el DSP TMS320DM6437 de Texas Instruments.

Figura 3.1: Aspecto de la tarjeta DM6437EVM.

3.2.1. Componentes principales

Los principales componentes que se incluyen en la tarjeta son los siguientes:

- Un DSP TMS320DM6437 de Texas Instruments funcionando a 600 MHz.
- Un descodificador de video TVP5146M2 son soporte para video compuesto y S-Video.
- 3 convertidores de video digital a analógico: vídeo en componentes, vídeo compuesto y vídeo RGB.
- 128 Mbytes de memoria RAM DDR2.
- Interfaces UART y CAN.
- 16 Mbytes de memoria Flash no volátil, 64 Mbytes de Flash NAND y 2 Mbytes de SRAM.
- Un *codec* de audio estéreo AIC33.
- Una interfaz I^2C.
- Una interfaz Ethernet a 10/100 Mbps.
- Interfaz de emulación JTAG integrada.
- 4 LEDs y 4 interruptores *micro-switches*.
- Fuente de alimentación única (+5V).
- Conectores de expansión para conectar una tarjeta auxiliar.
- Una interfaz VLYNQ.
- Una interfaz S/PDIF analógica y otra óptica.

3.2.2. Funcionamiento

Como se puede ver en la figura 3.2, el DSP TMS320DM6437 se conecta a los periféricos de la placa a través de interfaces de dedicadas integradas en el propio DSP y a través de un bus EMIF (External Memory InterFace) de 8 bits. La memoria DDR2 se conecta a través de un bus dedicado de 32 bits. El bus del EMIF se puede conectar a la memoria Flash, a la SRAM, a la NAND y a los conectores de expansión de la tarjeta auxiliar mediante los *jumpers* de la tarjeta.

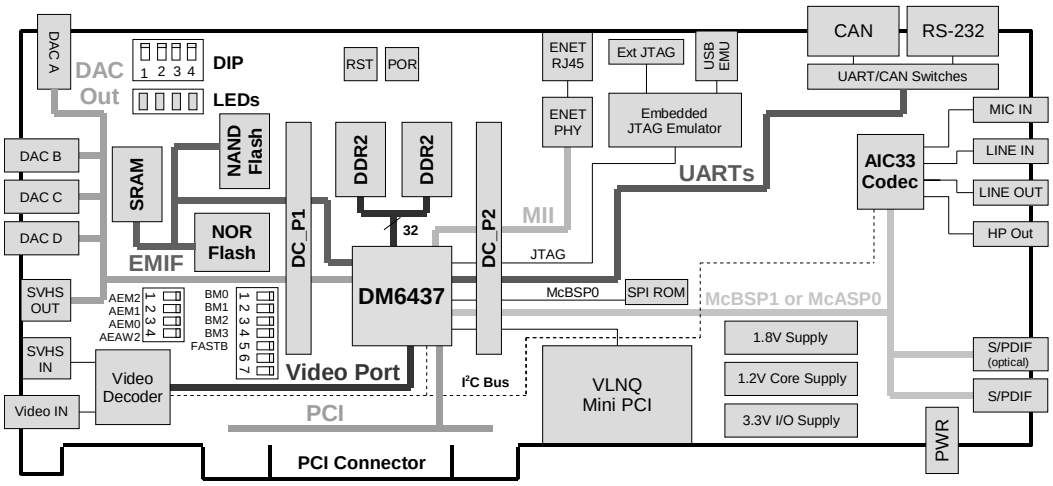


Figura 3.2: Diagrama de bloques de la tarjeta DM6437EVM. [23]

La placa dispone de chips convertidores de vídeo analógico a digital y de video digital a analógico que se conectan al DSP y permiten a éste manejar *streams* de video. Posee un decodificador TVP5146M2 y 4 convertidores digital a analógico (de los cuales 3 conectores están disponibles). Las funciones de visualización de información en pantalla (OSD, On Screen Display) se implementan por software en el DSP.

Un *codec* AIC33 permite al DSP transmitir y recibir señales de audio. El bus I^2C se usa como interfaz de control del *codec* mientras que el McBSP (*Multichannel Buffered Serial Port*), se utiliza para manejar el *stream* de audio. La conexión con la señal de audio se realiza mediante conectores tipo *jack* de 3.5 mm, que se corresponden con la entrada de micrófono, la entrada de línea, la salida de línea y la salida de los auriculares.

La tarjeta dispone de 4 LEDs y 4 interruptores de tipo *micro-switch* que se pueden usar como medio para interaccionar con la tarjeta. El acceso a estos dispositivos se realiza mediante expansores I^2C.

Las interfaces VLYNQ y EMAC (Ethernet Medium Access Controller) se integran como periféricos dentro del DSP. VLYNQ solamente está disponible cuando la interfaz PCI (Peripheral Component Interconnect) no está siendo utilizada.

Para alimentar la placa se utiliza una fuente de alimentación externa de 5V. Los reguladores conmutados integrados en la tarjeta proporcionan +1.2V al núcleo de la CPU, +3.3V a los periféricos y +1.8V a la memoria DDR2. La tarjeta se mantiene en estado de reset hasta que estas alimentaciones se estabilizan dentro del margen de funcionamiento.

Por último, el entorno de desarrollo integrado Code Composer se comunica con la tarjeta a través de un conector USB al emulador integrado en la propia placa o bien empleando un emulador externo que se conecta a la tarjeta mediante un conector JTAG de 14 pines.

3.2.3. Mapa de memoria

El DSP TMS320DM6437 tiene un espacio de direccionamiento muy amplio. El código del programa y los datos se pueden situar en cualquier parte del espacio de direcciones. Las direcciones pueden ser de múltiples tamaños, dependiendo de la implementación hardware.

En la tabla 3.1 se muestra el mapa de memoria que proporciona del fabricante de la tarjeta DM6437EVM. En la columna de la izquierda se muestran las direcciones del DSP y en la derecha se indica que dispositivos o interfaces están mapeados en esas direcciones. Por defecto, la memoria interna se sitúa al comienzo del espacio de direccionamiento (en la sección 3.3.2 se explica detalladamente la memoria interna del DSP).

Tabla 3.1: Mapa de memoria simplificado de la tarjeta DM5437EVM.

Dirección inicial	Región
0x1080000	Caché/RAM
0x4200000	CS2
0x4400000	CS3
0x4600000	CS4
0x4800000	CS5
0x4C00000	VLNQ
0x8000000	DDR

El DSP incorpora una interfaz EMIF dual. Una EMIF dedicada se conecta directamente a la memoria DDR2. La otra EMIF está situada en el espacio CS2 del mapa de memoria. La memoria Flash, la NAND o la SRAM se pueden mapear en la región CS2 mediante el *jumper* JP2.

3.2.4. Conectores

La tarjeta DM5437EVM se presenta en forma de PCB (Printed Circuit Board) de 210 x 115 mm y 10 capas. Posee 23 conectores que proporcionan conexiones con diferentes periféricos. En la figura 3.3 se puede observar la disposición de dichos conectores. En la tabla 3.2 se indican los tipos de conectores y la función de cada uno.

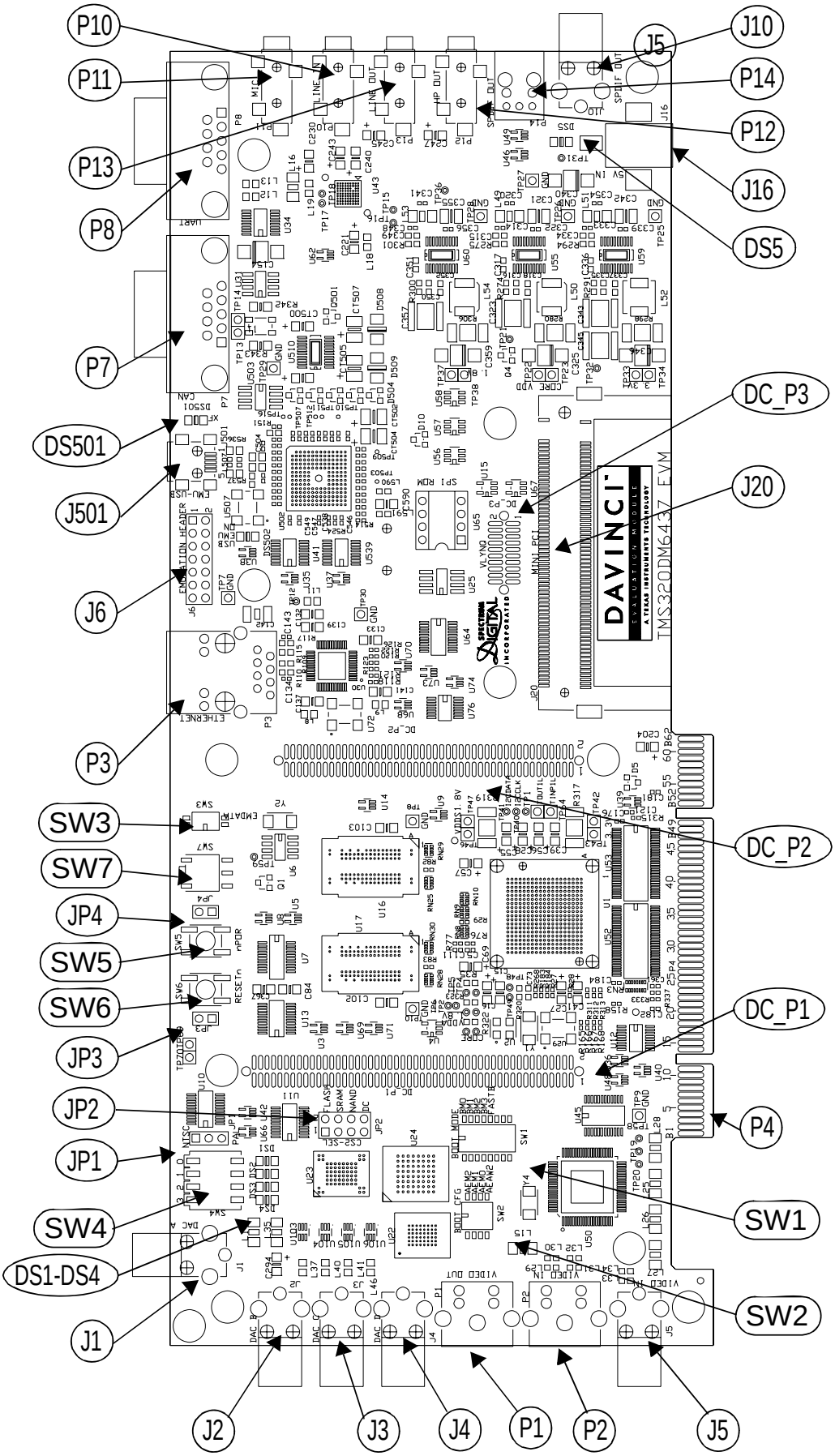


Figura 3.3: Conectores de la tarjeta DM6437EVM. [23]

Tabla 3.2: Tipo y función de los conectores de la tarjeta DM5437EVM.

Conector	Tipo	Función
J1	RCA	DAC A (No colocado)
J2	RCA	DAC B
J3	RCA	DAC C
J4	RCA	DAC D
J5	RCA	Entrada de video
J6	14	Conetor de emulación externa
J10	RCA	Salida S/PDIF
J16	2.5 mm	Entrada +5V
J20	2 x 62	Conector mini PCI
J501	Mini USB	Conector USB emulación integrada
P1	4 Pin DIN	Salida S-Video
P2	4 Pin DIN	Entrada S-Video
P3	RJ-45	Ethernet
P4	PCI	PCI
P7	9 Pin D-sub	CAN
P8	9 Pin D-sub	UART RS-232
P10	3.5 mm	Entrada de línea estéreo
P11	3.5 mm	Entrada de micrófono
P12	3.5 mm	Salida de auriculares
P13	3.5 mm	Salida de línea estéreo
P14	Óptico	Salida S/PDIF
DC_P1	2 x 50	Expansión
DC_P2	2 x 45	Expansión
DC_P3	2 x 10	Expansión

3.3. El DSP TMS320DM6437

Dentro de la gama de DSPs de la compañía Texas Instruments, los dispositivos TMS320 tienen una arquitectura diseñada específicamente para el procesado de señal en tiempo real. Dentro de este grupo se encuentra la familia TMS320C6000 destinada a aplicaciones multimedia que necesiten gran rendimiento y en las que el consumo energético no sea un requisito. Por último, a esta familia pertenece la subfamilia TMS320C64+ (también llamada DaVinci) a la cual pertenece el DSP utilizado en este proyecto.

3.3.1. Descripción general

El TMS320DM6437 es un DSP de alto rendimiento que posee una arquitectura de tipo VLIW (*Very Long Instruction Word*). A continuación se esquematizan sus características más relevantes:

- **Arquitectura de la CPU:**

  - VLIW con 8 unidades funcionales independientes (con aritmética de punto fijo):
    - 6 ALUs: cada una soporta una operación de 32 bits, dos de 16 bits o 4 de 8 bits en cada ciclo de reloj.
    - 2 multiplicadores: cada uno soporta dos multiplicaciones de 16 x 16 bits (cada resultado de 32 bits) o cuatro multiplicaciones de 8 x 8 bits (cada resultado de 16 bits) en cada ciclo de reloj.
  - Arquitectura segmentada (*pipelining*).
  - Arquitectura de memoria de tipo *Hardvard*.
  - Juego de instrucciones RISC.
  - Representación de los datos *little endian*.
  - Soporte para lectura y escritura de datos no alineados al tamaño de palabra (32 bits).
  - 64 registros de propósito general de 32 bits.
  - Empaquetado de instrucciones para reducir el tamaño del código.
  - Todas las instrucciones son condicionales en cuanto a su ejecución.
  - Controlador IDMA (*Internal Direct Memory Access*) para realizar transferencias de datos entre las memorias y los periféricos internos.

- **Rendimiento:**
  - Frecuencias de reloj de 400, 500, 600, 660 ó 700 MHz.
  - Periodos de instrucción de 2.5, 2, 1.67, 1.51 ó 1.43 ns.
  - Hasta 8 instrucciones de 32 bits en cada ciclo de reloj.
  - 3200, 4000, 4800, 5280 ó 5600 MIPS.

- **Jerarquía de memoria:**
  - Nivel 1 (L1):
    - Nivel 1 de programa (L1P) de 32 KBytes, configurables como memoria SRAM, caché o combinación de ambas.
    - Nivel 1 de datos (L1D) de 80 KBytes, de los cuales 48 KBytes son de memoria SRAM y 32 KBytes se pueden configurar como memoria SRAM, caché o combinación de ambas.
  - Nivel 2 (L2):
    - Nivel 2 de datos y programa de 128 KBytes, configurables como SRAM, caché o combinación de ambas.

- **Periféricos:**
  - Subsistema de procesado de vídeo VPSS (*Video Processing SubSystem*):
    - *Front End*:
      - Interfaz (de 8/16 bits) BT.601/BT.656 YC_bC_r 4:2:2.
      - Interfaz directa (*glueless*) con los chips descodificadores de vídeo comunes.
      - Módulo de auto-exposición, auto-balance del blanco y auto-enfoque.
      - Motor de redimensionamiento de imágenes desde 1/4x a 4x.
    - *Back End*:
      - Dispositivo *hardware* de OSD.
      - 4 DACs funcionando a 54 MHz que permiten obtener: vídeo compuesto NTSC/PAL, S-Vídeo o vídeo en componentes (YP_bP_r o RGB).
      - Salida digital: YUV de 8/16 bits o RGB de hasta 24 bits.
  - Interfaces de memoria externa EMIF (*External Memory Interface*).

    - Controlador de memoria DDR2 SDRAM de 32 bits con un espacio de direccionamiento de 256 MBytes.
    - EMIF asíncrono de 8 bits de ancho de bus (EMIFA).
  - Controlador de acceso directo a memoria EDMA (*Enhanced Direct Memory Access*) de 64 canales independientes.
  - 2 *timers* de propósito general de 64 bits (cada uno se puede configurar como dos *timers* de 32 bits).
  - Un *timer watch dog* de 64 bits.
  - 2 UARTs (una de ellas con control de flujo RTS y CTS).
  - Bus I^2C (*Inter-Integrated Circuit*) maestro/esclavo.
  - 2 puertos serie multicanal con *buffer* McBSP (*Multichannel Buffered Serial Port*).
  - Puerto serie de audio multicanal McASP0 (*Multichannel Audio Serial Port*).
  - Interfaz HPI (*Host Port InterfaceE*) de 16 bits.
  - Controlador HECC (*High-End CAN Controller*).
  - Interfaz PCI (*Peripheral Component Interconnect*) maestro/esclavo de 32 bits a 33 MHz y 3.3V.
  - EMAC (*Ethernet Medium Access Controller*) de 10/100 Mbps.
  - Interfaz VLYNQ.
  - 3 salidas PWM (*Pulse Width Modulator*).
  - *Bootloader* ROM integrado.
  - Modos de ahorro de energía individuales.
  - Generadores de reloj flexibles basados en PLLs.
  - Interfaz JTAG.
  - Hasta 111 pines de entrada-salida de propósito general GPIO (*General Purpose I/O*, multiplexados con otras funciones del DSP.

Por último, en la figura 3.4 se muestra un diagrama de bloques funcionales del DSP.

3.3.2. Jerarquía de memoria

Como se muestra en la figura 3.5, la memoria de la subfamilia C64x+ consta de dos niveles de memoria interna (L1 y L2) que se pueden configurar como SRAM, caché o como combinación de ambas. Además posee un tercer nivel de memoria externa (L3) con hasta 256 MBytes direccionables.

3.3.2.1. Mapa de memoria

En la tabla 3.3 se muestran los rangos más importantes del mapa de memoria del DSP indicándose la dirección origen y destino, el tamaño del rango y su utilización. Puesto que el DSP no soporta el EMIFA CS0 y CS1, no se muestra en la tabla.

3.3.2.2. L1P

La memoria interna de programa de nivel 1 (L1P) tiene un tamaño de 32 KBytes. Se puede configurar como memoria SRAM, caché de correspondencia directa o una combinación de ambas. Esta memoria se utiliza con el objetivo de maximizar el rendimiento de la ejecución del código.

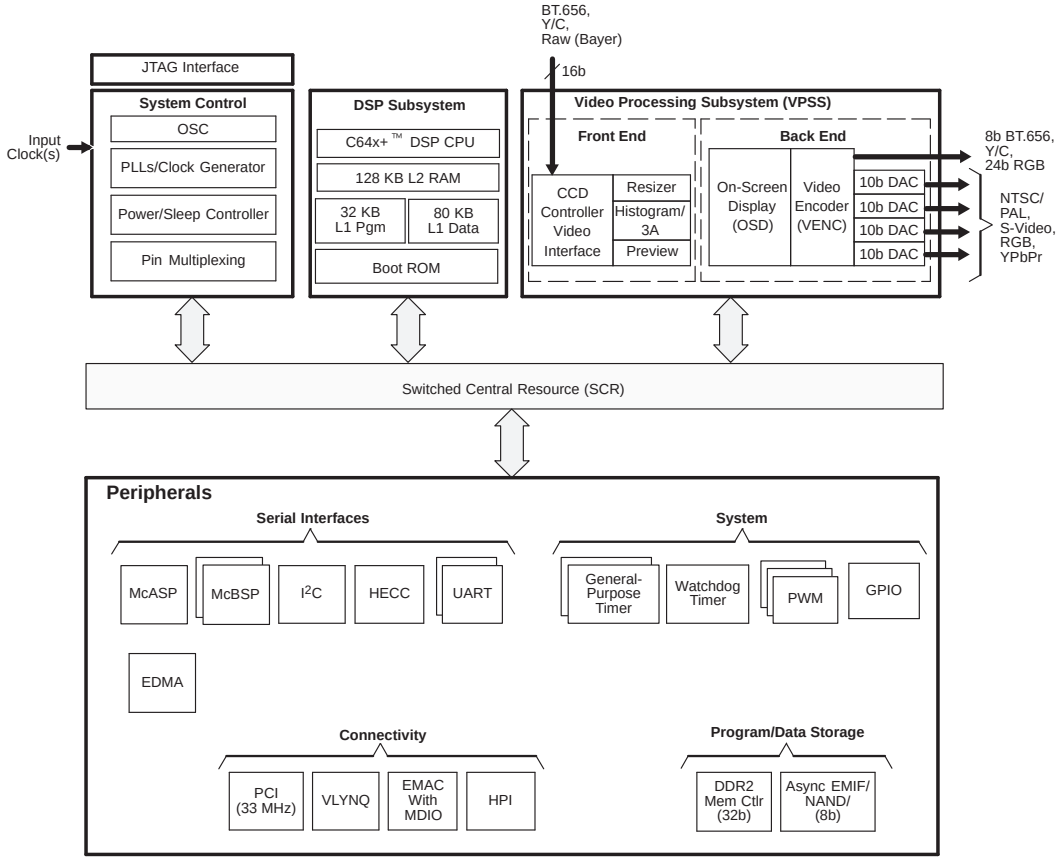


Figura 3.4: Diagrama de bloques funcionales del DSP TMS320DM6437. [28]

Los tamaños de la memoria caché que se pueden configurar son 4 KB, 8 KB, 16 KB y 32 KB. El resto de memoria disponible, si cabe, queda configurada como SRAM.

En la memoria caché se guardan las instrucciones del código del programa a medida que se van ejecutando. Las memorias que son accesibles a esta caché son la SRAM de nivel 2 y la memoria externa.

En la memoria SRAM también se almacenan instrucciones del código del programa con la diferencia de que es el programador quién elige que fragmento de código se guarda en esta memoria. La información de la SRAM L1P no se guarda en la caché L1D, en la caché L1P ni en la caché L2.

3.3.2.3. L1D

La memoria interna de datos de nivel 1 (L1D) tienen un tamaño de 80 KBytes. De ellos, 48 KBytes son de memoria SRAM y los 32 KBytes restantes se pueden configurar como SRAM, caché asociativa por conjuntos de dos vías o una combinación de ambas. La memoria L1D se utiliza para maximizar el rendimiento en el procesamiento de los datos.

Los tamaños de la memoria caché que se pueden configurar son 4 KB, 8 KB, 16 KB y 32 KB. Cuando queda memoria disponible, ésta queda configurada como SRAM.

En la memoria cache se guardan los datos que se leen o se escriben por la CPU. Las memorias que son accesibles a esta caché son la SRAM de nivel 2 y la memoria externa.

En la memoria SRAM también se almacenan datos del programa con la única diferencia de que es el programador quién elige que datos se guardan en ella. La información de la SRAM L1D

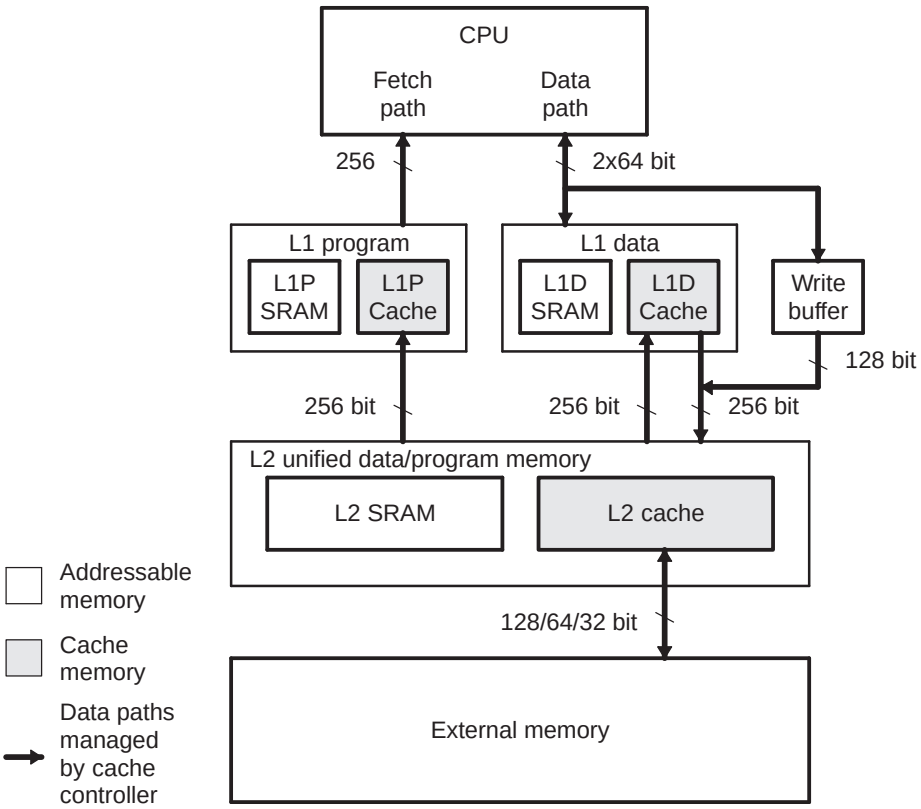


Figura 3.5: Arquitectura de memoria de la subfamilia C64x+. [29]

no se guarda en la cache L1D, en la caché L1P ni en la caché L2.

3.3.2.4. L2

La memoria interna de nivel 2 (L2) tiene un tamaño de 128 KBytes. Se puede configurar como memoria SRAM, caché asociativa por conjuntos de cuatro vías o una combinación de ambas. Ésta memoria está situada entre la memoria de nivel 1, que es más rápida, y la memoria externa, que es más lenta.

Los tamaños de la memoria caché que se pueden configurar son 32 KB, 64 KB y 128 KB. El resto de memoria, si cabe, queda configurada como SRAM.

En la memoria caché de este nivel se almacenan el código y los datos del programa. La única memoria que es accesible a la caché de nivel 2 es la memoria externa.

En la memoria SRAM también se almacenan los datos y el código del programa con la única diferencia de que es el programador quién elige que datos e instrucciones se guardan en ella. La información de la SRAM L2 no se guarda en la caché L2, pero si en la caché L1D y en la caché L1P.

3.4. El entorno de desarrollo integrado

El entorno de desarrollo que se ha usado para trabajar con la placa DM6437EVM es Code Composer Studio (CCS) en su versión 5.2.1.00018 [26]. Esta herramienta dispone de todas los elementos necesarios para desarrollar y depurar software en dispositivos empotrados. Algunas de las características más sobresalientes para el objetivo de este proyecto son:

- **Compilador C6000**. Soporte para C++ y C (C89). El soporte para C99 es casi completo a

Tabla 3.3: Resumen del mapa de memoria del DSP TMS320DM6437.

Dirección inicial	Dirección final	Tamaño	Función
0x0010 0000	0x0010 FFFF	64KB	Boot ROM
0x0080 0000	0x0081 FFFF	128KB	L2 RAM/Cache
0x00E0 8000	0x00E0 FFFF	32KB	L1P RAM/Cache
0x00F0 4000	0x00F0 FFFF	48KB	L1D RAM
0x00F1 0000	0x00F1 7FFF	32KB	L1D RAM/Cache
0x1010 0000	0x1010 FFFF	64KB	Boot ROM
0x1080 0000	0x1081 FFFF	128KB	L2 RAM/Cache
0x10E0 8000	0x10E0 FFFF	32KB	L1P RAM/Cache
0x10F0 4000	0x10F0 FFFF	48KB	L1D RAM
0x10F1 0000	0x10F1 7FFF	32KB	L1D RAM/Cache
0x4200 0000	0x42FF FFFF	16MB	EMIFA Data (CS2)
0x4400 0000	0x44FF FFFF	16MB	EMIFA Data (CS3)
0x4600 0000	0x46FF FFFF	16MB	EMIFA Data (CS4)
0x4800 0000	0x48FF FFFF	16MB	EMIFA Data (CS5)
0x4C00 0000	0x4FFF FFFF	64MB	VLYNQ (Remote data)
0x8000 0000	0x8FFF FFFF	256MB	DDR2 Memory Controller

partir de la versión 7.4 del compilador [25] (versión que se usa en este libro).

- **DSP/BIOS**. Es un sistema operativo en tiempo real usado en una amplia variedad de procesadores empotrados. Incorpora funciones para la gestión y sincronización de tareas, configuración de la memoria (ver sección 4.2.3), acceso a periféricos de bajo nivel, etc. Actualmente se ha sustituido por SYS/BIOS.

- ***Profiler***. Su función es analizar el rendimiento del software. Como se muestra en la sección 4.5, en este proyecto se usa para medir el número de ciclos del procesador empleados en la ejecución de cada función.

- ***Scripting***. El lenguaje de *scriping* de CCS es *javascript*. Con él se pueden automatizar tareas repetitivas sin la intervención del usuario. En este proyecto se usa para realizar, de forma automática, *profiles* y medidas en la placa para una lista de secuencias de prueba (ver secciones 4.4.2 y 4.5.1).

- **Múltiples objetivos**. Para cada proyecto de software se pueden crear múltiples objetivos de compilación. Es muy útil para compilar un mismo código en varias arquitecturas diferentes (ver sección 4.2.1).

Por último, CCS permite ejecutar el código compilado en un simulador o bien, mediante un emulador, transferirlo a una tarjeta de desarrollo que integre un DSP para posteriormente ser ejecutado en este último. En la sección 4.2.1 se detalla cómo se han usado las dos opciones en este proyecto.

Capítulo 4

Desarrollo

En este capítulo se presentan los resultados de este proyecto. La sección 4.1 presenta una breve historia del descodificador openHEVC y define su arquitectura básica.

En la sección 4.2 se describe detalladamente el proceso de migración de openHEVC al procesador digital de señales. En la sección 4.4 se presentan las medidas de velocidad de descodificación obtenidas para el DSP y en la sección 4.5 se detalla el porcentaje de carga computacional requerido por cada bloque del descodificador.

Por último, en la sección 4.6 se comparan los resultados de este trabajo con los obtenidos en otros trabajos similares.

4.1. Descodificador openHEVC

Durante el proceso de estandarización de HEVC por el grupo JCT-VC, Guillaume Martres desarrolló una implementación inicial de un descodificador HEVC para imágenes *intra* que presentó en Google Summer of Code (GSoC) [12]. Este descodificador se realizó para integrarse en Libav [2], un conjunto de bibliotecas portables para manejar formatos multimedia.

Basándose en este trabajo, se creó el proyecto de software libre openHEVC como un *fork*[1] de Libav. Actualmente este proyecto se desarrolla activamente en *GitHub*[2] [3] y tiene como objetivo servir de investigación [11], sobre todo en temas relacionados con la optimización del rendimiento de plataformas multi-núcleo para la descodificación en tiempo real de UDTV (*Ultra High Definition Television* —4096x2048 píxeles—).

Por último, durante el desarrollo de esta memoria, se ha creado un *fork* de openHEVC [6] con el objetivo de migrar el descodificador al procesador digital de señal TMS320DM6437 de Texas

Figura 4.1: Cronología de openHEVC.

[1] En terminología Git, un fork es una bifurcación de un proyecto de software libre.
[2] GitHub es una plataforma para el desarrollo colaborativo de software usando el sistema de control de versiones Git.

Instruments. La versión de openHEVC que se ha usado para la migración es la del 18 de marzo de 2013.

4.1.1. Arquitectura software

En la figura 4.2 se muestra la arquitectura *software* simplificada del proceso de descodificación de una *slice* en openHEVC.

En la función `hls_slice_data` se descodifican todos las CTUs del *slice*. Para ello, se llama a la función recursiva `hls_coding_tree` donde se desciende hasta el nivel de las CUs. Para cada CU, la función `hls_prediction_unit` computa la predicción *inter* o *intra*. Por otra parte, la función `hls_transform_tree` busca cada una de las TUs para posteriormente ser descodificadas por `hls_transform_unit`. Tras descodificar todas las CTUs se aplican el filtro antibloques (`hls_deblocking_filter`) y el filtro SAO (`hls_sao_filter`) si están habilitados.

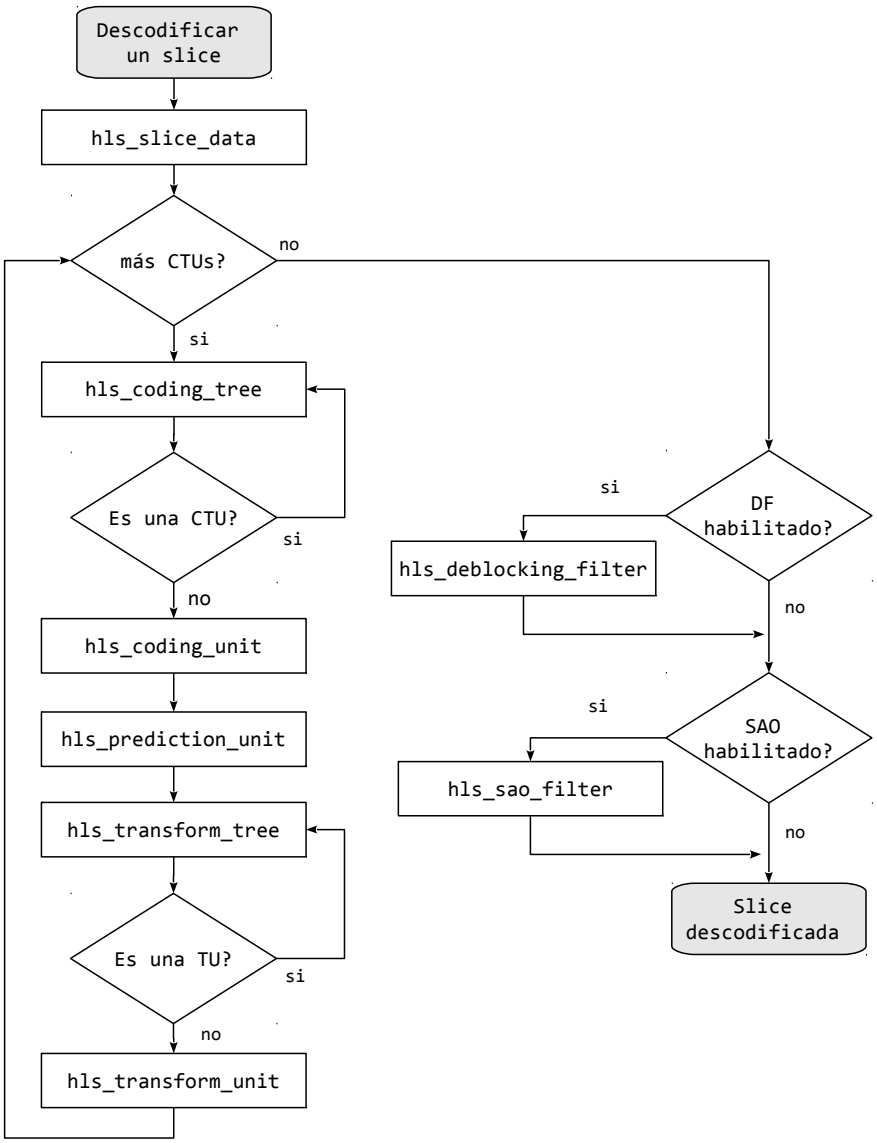


Figura 4.2: Diagrama de flujo simplificado del proceso de descodificación de una *slice* en openHEVC.

4.1.2. Herramintas de compilación y desarrollo

OpenHEVC es un proyecto de *software open source* en activo desarrollo en GitHub. GitHub es una plataforma web donde es posible alojar el código fuente de un proyecto de *software* y mediante el uso del sistema de control de versiones Git varios programadores trabajar colaborativamente en él. A continuación se describe el proceso de obtención del código fuente de openHEVC, su compilación y finalmente la forma de utilizar el entorno desarrollo integrado Eclipse, en una computadora con el sistema operativo GNU/Linux.

En primer lugar es necesario disponer de algunas bibliotecas, de las cuales depende openHEVC, en la plataforma donde se compilará el código. Las más importantes son `libyasm` y `libsdl`. La primera se trata de un ensamblador para las arquitecturas x86 y AMD64. Por otra parte, *libsdl* es una biblioteca que se utiliza en este proyecto para visualizar por pantalla el video descodificado.

A continuación, en una terminal de comandos se puede iniciar la descarga de la última versión disponible de openHEVC mediante la herramienta Git:

```
$ git clone git://github.com/OpenHEVC/openHEVC.git
```

Posteriormente, dentro del directorio del proyecto, se selecciona la rama (*branch*[3]) de trabajo `hm10.0_au`. Esta es la rama más activa y donde se desarrolla el descodificador compatible con el software de referencia HM10.0.

```
$ cd openHEVC
$ git checkout hm10.0_au
```

A continuación se crea el directorio donde se compilará el *software* y se accede al él:

```
$ mkdir build
$ cd build
```

CMake es la herramienta de automatización de código que se utiliza en openHEVC. Con CMake se pueden generar *makefiles*[4] nativos y proyectos para IDEs de manera sencilla. Con la siguiente orden se generan los *makefiles UNIX* compatibles con las herramientas de compilación de GNU:

```
$ cmake -DCMAKE_BUILD_TYPE=RELEASE ..
```

Una vez que se han obtenido los *makefiles*, se debe utilizar la herramienta GNU Make para iniciar el proceso de compilación del *software*:

[3] En terminología Git, una *branch* o rama es una bifurcación del código dentro de un mismo proyecto.

[4] Los *makefiles* son archivos donde se definen un conjunto de instrucciones utilizadas para compilar un proyecto de software.

```
$ make
```

Opcionalmente, tras la compilación se pueden instalar los ejecutables en el sistema de la siguiente manera:

```
$ make install
```

Tras haber obtenido el ejecutable la siguiente orden permite iniciar la descodificación de las imágenes en una secuencia de prueba:

```
$ ./hevc -i secuencia_de_prueba.yuv
```

Por último, para poder trabajar sobre el código, se puede utilizar la herramienta CMake para crear un proyecto para el entorno de desarrollo integrado Eclipse CDT:

```
$ cmake -DCMAKE_BUILD_TYPE=DEBUG -G "Eclipse CDT4 - Unix Makefiles" ..
```

Como resultado, en el directorio *build* se ha creado el fichero de proyecto (`.project`) que puede ser importado dentro del entorno gráfico de Eclipse mediante la orden File->Import... Tras crear el proyecto se pueden editar, compilar y depurar el código con las facilidades que incorpora Eclipse.

Actualmente la herramienta CMake no pude generar proyectos para el entorno Code Composer Studio (CCS). Por este motivo, para trabajar en este entorno, es necesario crearlos manualmente tal y como se ha realizado en la siguiente sección.

4.2. Migración de openHEVC al DSP

Puesto que openHEVC está pensado para ejecutarse en plataformas PC ha sido necesario migrar el código a la placa DSP DM6437EVM. En las siguientes subsecciones se detalla el proceso seguido para realizar dicha migración.

4.2.1. Proyecto CCS

En la figura 4.3 se presenta el esquema del proyecto CCS. Se trata de un proyecto para el sistema operativo DSP/BIOS donde se definen dos objetivos:

1. Emulador. Se usa para transferir el programa ejecutable a la placa DM6437EVM (ver sección 4.4).
2. Simulador. Usado principalmente para ejecutar el código en un simulador y obtener los datos de rendimiento (ver sección 4.5).

Las configuraciones del emulador y el simulador se describen en los ficheros EVMDM6437.ccxml y TMS320DM6437.ccxml respectivamente.

En la tabla 4.1 se muestran las principales opciones definidas en cada uno de los objetivos. El modelo de depuración *Full symbolic debug* (DWARF[5]) permite que el código sea depurable y

[5]DWARF es un estándar de depuración ampliamente usado (`http://dwarfstd.org`).

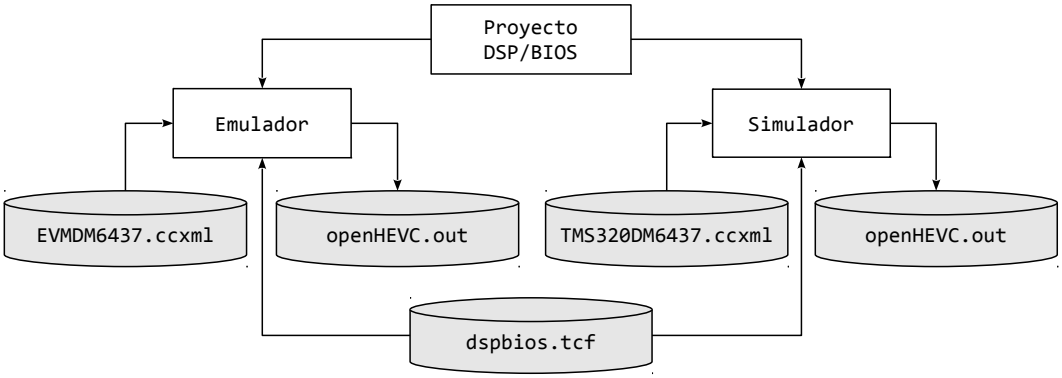


Figura 4.3: Esquema del proyecto CCS.

analizable con la herramienta *profiler* [32]. En la simulación, la opción *auto inline* se ha configurado a cero con el fin de que el compilador no cree funciones *inline*[6] de manera automática y por lo tanto todas las funciones del código sean mesurables por el *profiler*.

Tabla 4.1: Principales opciones del proyecto CCS.

Opción	Emulador	Simulador
Dispositivo	EVMDM6437	TMS320DM6437
Conexión	DSK-EVM-eZdsp onboard USB Emulator	...
DSP/BIOS	5.41.13.42	5.41.13.42
Modelo depuración:	none	DWARF
Optimización	O3	O3
Auto inline	...	0
Defines	EMULATOR	...
Conf. objetivo	EVMDM6437.ccxml	TMS320DM6437.ccxml
Conf. DSP/BIOS	dspbios.tcf	dspbios.tcf

4.2.2. Configuración del simulador

Para este proyecto se ha usado el simulador con tiempos *DM6437 Device Cycle Accurate Simulator*. Se trata de un simulador que, a diferencia de los simuladores funcionales, tiene en cuenta el tiempo invertido en las transferencias DMA y la carga que añaden los accesos a memoria externa y los fallos de cachés. Por este motivo su comportamiento es idéntico a la ejecución en el *hardware* real.

Para poder realizar una simulación lo más fiel posible a la ejecución real sobre el DSP, se ha tenido que indicar al simulador las características de la CPU simulada. Estás características se definen en el archivo TMS320DM6437.ccxml y se muestran en la tabla 4.2.

[6] En C, una función definida como *inline* hace que el compilador inserte el cuerpo de dicha función en cada punto donde es llamada.

Tabla 4.2: Características de la CPU simulada definidas en TMS320DM6437.ccxml.

Opción	Valor
Dispositivo	DM6437
Tipo de simulador	Cycle Accurate
Reloj CPU (MHz)	600
Reloj EMIF (MHz)	162
Modo *narrow* (EMIF)	Deshabilitado
Endianess	Little Endian

4.2.3. Configuración de DSP/BIOS

El objetivo del archivo dspbios.tcf es describir la configuración del sistema operativo DSP/BIOS. Las únicas opciones que se han configurado son las que tienen que tienen alguna relación con la memoria del DSP. En la tabla 4.3 se muestra un resumen de la configuración realizada.

Tabla 4.3: Configuración de la memoria en el sistema operativo DSP/BIOS.

Memoria	Tamaño	Espacio
L1P	32KB	Caché de nivel 1
L1D	32KB	Caché de nivel 1
L2	128KB	Caché de nivel 2
L3	128MB	Código/Datos cacheables
SRAM externa	2MB	Código/Datos

Para aumentar la velocidad de ejecución del código se ha usado la máxima cantidad de memoria caché disponible en los niveles L1 y L2. También, con el mismo propósito, se ha configurado toda la memoria externa como cacheable con el fin de que no sea necesario realizar accesos de larga distancia a la memoria cada vez que se quiera leer o escribir en ella [30]. Para realizar esto último, se han configurado los registros MAR (*Memory Attribute Register*) de la CPU mediante la herramienta gráfica *Configuration Tool*[7].

4.2.4. Modificaciones del código

Para compilar y ejecutar correctamente el código de openHEVC[8] en el DSP, ha sido necesario realizar modificaciones en varios archivos fuente. A continuación se detallan estos cambios:

config.h

En este archivo de cabecera se definen una serie de *flags* para indicar que características dispone la plataforma donde se ejecutará el código. Cuando la plataforma para la que se compila es un PC, este archivo se genera automáticamente con el *script* `configure.sh` de Libav. Para adaptarlo al DSP ha sido necesario modificar manualmente las líneas indicadas en el listado 4.1.

A continuación se detalla el significado de cada uno de esos *flags*:

[7] Disponible en la instalación de Code Composer Studio.
[8] Disponible en `http://github.com/OpenHEVC/OpenHEVC`.

Listado 4.1: Cambios realizados a config.h.

```
#define ARCH_X86 0
#define ARCH_X86_64 0
#define CONFIG_MEMALIGN_HACK 1
#define HAVE_FAST_UNALIGNED 0
#define HAVE_INLINE_ASM 0
#define HAVE_ISATTY 0
#define HAVE_LLRINT 0
#define HAVE_LLRINTF 0
#define HAVE_MMX 0
#define HAVE_POSIX_MEMALIGN 0
#define HAVE_PTHREADS 0
#define HAVE_SYSCTL 0
#define HAVE_THREADS 0
#define HAVE_UNISTD_H 0
```

- **ARCH_X86 y ARCH_X86_64.** Indican que el microprocesador que ejecutará el código dispone de los conjuntos de instrucciones x86 y x86-64 respectivamente. Estos flags se han anulado ya que el DSP TMS320DM6437 no dispone de dichos juegos de instrucciones.
- **CONFIG_MEMALIGN_HACK.** En Libav, la función `av_malloc` se utiliza para reservar memoria dinámica alineada en direcciones múltiplos de 32. Para ello, puede hacer uso de la función `memalign`, `posix_memalign` o bien `_aligned_malloc`. En plataformas que no cuenten con ninguna de estas funciones, como es el caso del DSP, CONFIG_MEMALIGN_HACK permite emular a `memalign`.
- **HAVE_FAST_UNALIGNED.** Si esta macro está activa, se realizarán lecturas de datos potencialmente no alineados usando punteros a tipos `union`. En caso contrario se utilizará código específico para los compiladores GNU. Si no existe soporte especifico del compilador, las lecturas serán byte a byte. Se ha optado por esta última alternativa que, aunque es menos eficiente, genera menos problemas en la plataforma DSP.
- **HAVE_PTHREADS.** Esta macro indica que se dispone de *POSIX Threads*, una API para la creación de hilos de ejecución según el estándar POSIX. Los compiladores de Texas Instruments no disponen de dicha API y por lo tanto se ha desactivado.
- **HAVE_INLINE_ASM.** Se utiliza para optimizar la ejecución de ciertas funciones en `cabac.h` y `mathops.h` usando para ello instrucciones en ensamblador para la arquitectura x86. Puesto que el DSP no posee esta arquitectura, se ha deshabilitado.
- **HAVE_ISATTY.** Indica que se dispone de una terminal TTY de Unix y es utilizado en `log.c` para decidir la forma de colorear el texto de los mensajes de *logs*. Se ha deshabilitado puesto que el desarrollo de este trabajo se ha realizado en un sistema Windows sin terminal TTY.
- **HAVE_LLRINT y HAVE_LLRINTF.** Indica que se dispone de las funciones de redondeo `llrint` y `llrintf` respectivamente. En este caso, la biblioteca math.h del compilador C6000 no dispone de tales funciones y por lo tanto se ha deshabilitado su uso.
- **HAVE_POSIX_MEMALIGN.** Indica que la plataforma dispone de la función `posix_memalign` que se usa para reservar memoria dinámica alineada.
- **HAVE_SYSCTL.** La función `sysctl` permite modificar parámetros del núcleo Linux en tiempo de ejecución. Se utiliza en `pthread.c` durante la creación de los hilos de ejecución. En la plataforma DSP no se utiliza el núcleo Linux ni los hilos de ejecución POSIX y por lo tanto esta macro se ha deshabilitado.
- **HAVE_THREADS.** Para la migración del software a este DSP no es necesario hacer uso de la arquitectura multihilo de Libav y por lo tanto este *flag* se configura a cero.
- **HAVE_UNISTD_H.** El archivo de cabecera `unistd.h` se utiliza para acceder a la API de los sistemas operativos compatibles con POSIX. En el DSP no se necesita y se ha desactivado.

decana.c y decana.h

Se han creado estos dos archivos. En ellos se definen una serie de funciones para leer información de un fichero. Se utiliza como mecanismo para pasar información a la función `main.c` desde los programas de automatización de medidas que se describen en las secciones 4.4.2 y 4.5.1.

main.c

Se ha deshabilitado la suma de comprobación MD5 puesto ya que no era necesaria para la descodificación y consumía el 15% de la carga computacional. La función `libOpenHevcSetCheckMD5(MD5_DISABLE)` facilita enormemente esta tarea.

También se ha escrito la función `yuv_save` (ver listado 4.2). Permite guardar en el disco duro del PC la secuencia descodificada en formato YUV con el nombre `filename` que se le pasa como parámetro.

Listado 4.2: Función yuv_save.

```
void yuv_save(unsigned char *Y, unsigned char *U, unsigned char *V,
              int *wrap, int xsize, int ysize, char *filename) {
  FILE *f;
  int i;
  static int primera_apertura = 1;

  if(primera_apertura) {
    f = fopen(filename, "wb");
    fclose(f);
    primera_apertura = 0;
  }
  f = fopen(filename, "ab");
  if (f == NULL) {
    printf("It is not possible to write in the output file\n");
    exit(-1);
  }
  for(i=0; i<ysize; i++) {
    fwrite(Y + i * wrap[0], 1, xsize, f);
  }
  for(i=0; i<ysize/2; i++) {
    fwrite(U + i * wrap[1], 1, xsize/2, f);
  }
  for(i=0; i<ysize/2; i++) {
    fwrite(V + i * wrap[2], 1, xsize/2, f);
  }
  fclose(f);
}
```

openHevcWrapper.c

Se ha escrito el código necesario para contar los ciclos CPU que se gastan en ejecutar la función `avcodec_decode_video2`. Estas modificaciones se describen en la sección 4.4.1.

libavcodec/allcodecs.c

En este DSP una variable no inicializada explícitamente contiene un valor arbitrario. Por lo tanto a la variable `initialized` de la función `avcodec_register_all` se le ha asignado el valor 0.

libavcodec/utils.c

Se han inicializado las variables globales a un valor por defecto como se muestra en el listado 4.3. En el programa se espera que estas variables tengan este valor al ejecutarse por primera vez. Si no se inicializan explícitamente, estas variables contendrán valores no deseados.

libavutil/eval.c

Se han definido las constantes e y π tal y como se muestra en el listado 4.4. En el código, estas macros se utilizan, sin embargo no estaban definidas.

Listado 4.3: Inicialización de variables globales en utils.c.

```
1 static int volatile entangled_thread_counter = 0;
2 static int (*ff_lockmgr_cb)(void **mutex, enum AVLockOp op) = NULL;
3 static void *codec_mutex = NULL;
4 static void *avformat_mutex = NULL;
```

Listado 4.4: Definición de constantes e y π.

```
1 #define M_E         2.71828182845904523536028747135266250
2 #define M_PI        3.1416
```

libavutil/intreadwrite.h

En el programa existen funciones tales como `get_bits` que se utilizan para leer información de las NAL almacenadas en memoria. En estas lecturas, los datos no tienen por qué están alineados en direcciones de memoria múltiplo de 2.

El código para lecturas no alineadas optimizado para PC no funciona en el DSP. En su lugar, en esta última plataforma existen funciones intrínsecas como `_memXX()` y `_memXX_const()` que se utilizan para los accesos a memoria no alineada [31].

En este trabajo se ha optado por sustituir el código optimizado para PC por un código no optimizado que funciona correctamente en el DSP tal y como se muestra en el listado 4.5.

Listado 4.5: Código para lecturas no alineadas.

```
1 /* # define AV_RB32(p)    AV_RB(32, p) */
2 #    define AV_RB32(x)                                \
3     (((uint32_t)((const uint8_t*)(x))[0] << 24) |    \
4                 (((const uint8_t*)(x))[1] << 16) |    \
5                 (((const uint8_t*)(x))[2] <<  8) |    \
6                  ((const uint8_t*)(x))[3])
```

libavutil/libm.h

La función `rint` no está definida en el archivo de cabecera `math.h` del compilador C6000. Por lo tanto se han cambiado las llamadas a la función `rint` por la función `round`. Ambas funciones redondean un `double` y se pueden usar de forma equivalente.

libavutil/log.h

En el código del descodificador existen funciones para mostrar por pantalla *logs* que son de utilidad para el desarrollador. La ejecución de estas funciones no es importante para la funcionalidad del descodificador y pueden ralentizar su ejecución, por ello se han desactivado. En el listado 4.6 se muestra la manera en la que se ha anulado estas funciones[9]

libavutil/opt.c

Se han incluido las cabeceras `internal.h` (API interna de Libav) y `libm.h` (reemplazo de funciones matemáticas estándar de C) que faltaban. También se han definido las constantes del listado 4.4.

[9] Ésta técnica se encuentra descrita en http://stackoverflow.com/questions/7711570/disable-functions-using-macros.

Listado 4.6: Anular el uso de funciones de *log*.

```
#ifdef NOLOG
  #define av_log (void)sizeof
  #define av_vlog (void)sizeof
#else
  void av_log(void *avcl, int level, const char *fmt, ...) av_printf_format(3, 4);
  void av_vlog(void *avcl, int level, const char *fmt, va_list);
#endif
```

4.3. Secuencias

En las tablas 4.4 y 4.5 se presentan las secuencias que se han analizado en este trabajo. Se coreresponden con todas las secuencias de clase C (832x480) y D (416x240) recomendadas por JCT-VC [4]. Todas con una profundidad de color de 8 bis.

Los resultados se han limitado a dos parámetros de cuantificación QP (27 y 32) y 4 configuraciones: *All Intra*, *Low Delay P*, *Low Delay B* y *Random Access*. Por lo tanto se han analizado 40 secuencias de clase C y 32 de clase D.

Tabla 4.4: Secuencias de clase C.

Secuencia	Abreviatura	Imágenes	*Frame rate*
BasketballDrill	BD	500	50
BasketballDrillText	BT	500	50
BQMall	BM	600	60
PartyScene	PS	500	50
RaceHorses	RH	300	30

Tabla 4.5: Secuencias de clase D.

Secuencia	Abreviatura	Imágenes	*Frame rate*
BasketballPass	BP	500	50
BlowingBubbles	BB	500	50
BQSquare	BS	600	60
RaceHorses	RH	300	30

En las secuencias *Low Delay P* y *Low Delay B* no hay refresco *intra*. Sólo hay una imagen I, la primera de la secuencia. El resto son imágenes P y B respectivamente.

En las secuencias *Random Access* el periodo de imágenes I depende del *frame-rate* de la secuencia. Es 32 para 30fps, 64 para 60fps, etc. Es decir, se inserta una imagen I cada segundo. El resto son imágenes B.

4.4. Velocidad de descodificación

En este apartado se van a realizar las primeras medidas del rendimiento. En concreto, se va a medir la velocidad de descodificación, en términos de imágenes por segundo (FPS), en el hardware usando el emulador.

4.4.1. Forma de medir

El método para medir consiste en capturar, al inicio y al final del fragmento de código que se desea medir, el valor de un *timer*[10] que se incrementa en cada ciclo de reloj, mediante una llamada al sistema operativo. La resta de ambos valores (n_c) nos indica el número de ciclos que tarda el DSP en ejecutar el dicho fragmento de código. Para calcular la velocidad de descodificación se aplica la siguiente fórmula:

$$FPS = \frac{f n_i}{n_c} \tag{4.1}$$

donde f es la frecuencia del reloj del DSP (en Hz) y n_i es el número de imágenes que se han descodificado.

Con el objetivo de medir puramente el proceso de descodificación, la función que se ha medido es `avcodec_decode_video2`. Esta función se encarga únicamente de descodificar una NAL almacenada en memoria. Por lo tanto no lee la NAL del *bitstream* ni escribe las imágenes descodificadas en un archivo.

4.4.2. Automatización

Dada la gran cantidad de secuencias y el hecho de que el proceso de medición es una tarea mecánica, en esta sección se plantea una forma de automatizar el proceso, la cual ha sido esquematizada en la figura 4.4.

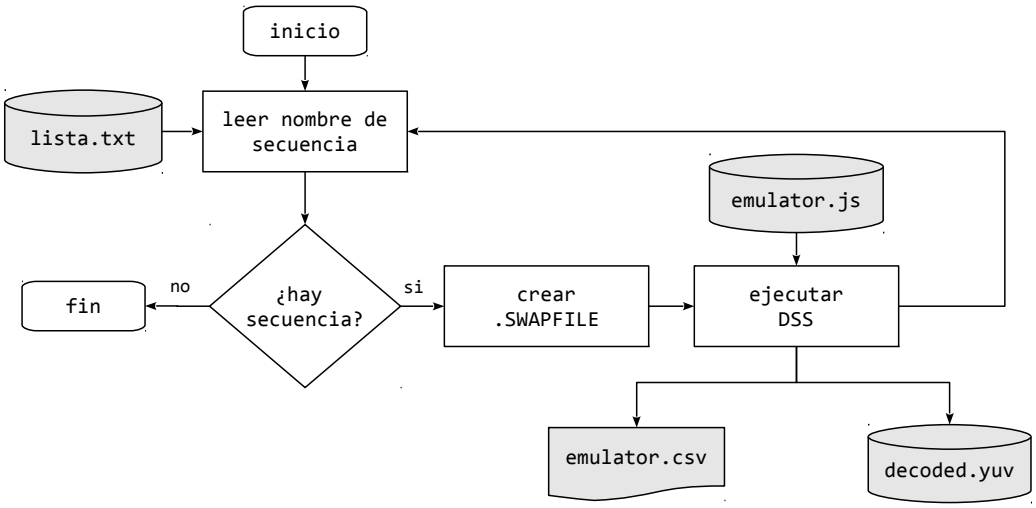


Figura 4.4: Esquema de la toma automática de medidas en la placa.

En primer lugar se dispone del archivo list.txt (ver listado 4.7), donde se definen los nombres de las secuencias que se quieren descodificar.

Por medio de un programa escrito en Python[11] se lee el nombre de una secuencia de este fichero y se escribe el archivo .SWAPFILE (ver listado 4.8). El fichero .SWAPFILE sirve como medio para pasar información al programa principal (`main.c`). En concreto, la información que se representa en cada línea de este fichero es:

1. Ruta del fichero de la secuencia.
2. Ruta del fichero con la secuencia descodificada.

[10] Es un periférico interno del DSP.
[11] Python es un lenguaje de programación interpretado y multiplataforma (`http://www.python.org`).

Listado 4.7: Archivo list.txt

```
BQMall_832x480_60_qp27.bin
BQMall_832x480_60_qp32.bin
BQSquare_416x240_60_qp27.bin
BQSquare_416x240_60_qp32.bin
BasketballDrillText_832x480_50_qp27.bin
BasketballDrillText_832x480_50_qp32.bin
BasketballDrill_832x480_50_qp27.bin
BasketballDrill_832x480_50_qp32.bin
BasketballPass_416x240_50_qp27.bin
BasketballPass_416x240_50_qp32.bin
PartyScene_832x480_50_qp27.bin
PartyScene_832x480_50_qp32.bin
RaceHorses_416x240_30_qp27.bin
RaceHorses_416x240_30_qp32.bin
RaceHorses_832x480_30_qp27.bin
RaceHorses_832x480_30_qp32.bin
BlowingBubbles_416x240_50_qp27.bin
BlowingBubbles_416x240_50_qp32.bin
```

3. Número de imágenes a descodificar.

4. Ruta del fichero donde se guardarán las medidas realizadas.

Listado 4.8: Ejemplo de archivo .SWAPFILE para una secuencia.

```
C:/automate/bitstreams/i_main/BQMall_832x480_60_qp27.bin
C:/automate/decoded/emulator/i_main/BQMall_832x480_60_qp27.yuv
100
C:/automate/emulator/emulator/emulator_i_main.txt
```

Posteriormente, este programa invoca al Debug Scripting Server (DSS). Como se explicó al comienzo de la sección 4.2, DSS nos permite automatizar, desde un *script*, cualquier tarea que se pueda realizar con el depurador de CSS. En este caso, en el *script* emulator.js tan sólo se indica al depurador que ejecute el programa openHEVC y espere a su finalización.

Como resultado de la ejecución de openHEVC, se han añadido al fichero emulator.txt las medidas realizadas. También se obtiene un archivo YUV con la secuencia descodificada.

El listado 4.9 representa un ejemplo del fichero emulator.txt para secuencias *All Intra*. Cada línea está compuesta por tres campos separados entre sí por espacios en blanco y ordenados de la siguiente manera:

1. Ruta de la secuencia medida.

2. Número de imágenes descodificadas.

3. Número de ciclos de reloj totales medidos.

Listado 4.9: Fichero emulator_i_main.txt para las cuatro primeras secuencias.

```
C:/automate/bitstreams/i_main/BQMall_832x480_60_qp27.bin 100 13119713997
C:/automate/bitstreams/i_main/BQMall_832x480_60_qp32.bin 100 10850148587
C:/automate/bitstreams/i_main/BQSquare_416x240_60_qp27.bin 100 4387291881
C:/automate/bitstreams/i_main/BQSquare_416x240_60_qp32.bin 100 3555984885
```

Con estos resultados se puede obtener la velocidad de descodificación mediante la ecuación 4.1. También se pueden graficar los resultados (ver sección 4.4.3).

Todos este proceso se repite para cada secuencia del fichero list.txt y para cada configuración (*All Intra, Low Delay P, Low Delay B* y *Random Access*).

4.4.3. Resultados

En este apartado se muestran los resultados, expresados en FPS, obtenidos aplicando la metodología explicada en las anteriores secciones. Las medidas se han realizado tomando las 100 primeras imágenes de cada secuencia.

Si comparamos las figuras 4.5 y 4.6 podemos apreciar que la velocidad de descodificación de las secuencias con QP 27 es aproximadamente un 20 % menor que en las secuencias con QP 32. Esto se debe a que, en las primeras, el *bit-rate* es mayor y por lo tanto es necesario descodificar mas información. Obviamente sucede lo mismo en las secuencias de clase D (figuras 4.7 y 4.8).

Los resultados son consistentes con el hecho de que, por lo general, la mayor velocidad de descodificación se consigue con las secuencias *Low Delay P*. Esto se debe a que es la única configuración cada imagen tiene una única referencia a una imagen anterior.

La descodificación de secuencias *All Intra* es más lenta debido a su alto *bit-rate* y al hecho de que carecen de predicción temporal.

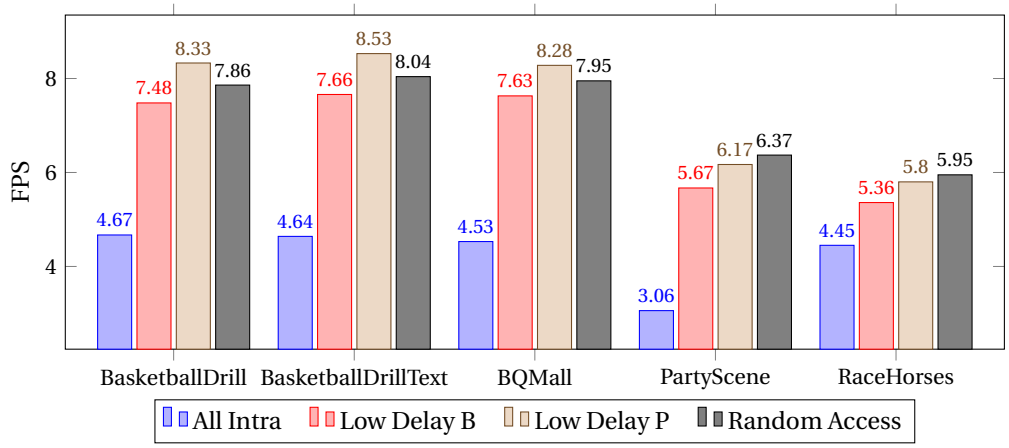


Figura 4.5: Velocidad de descodificación para secuencias de Clase C y QP 27.

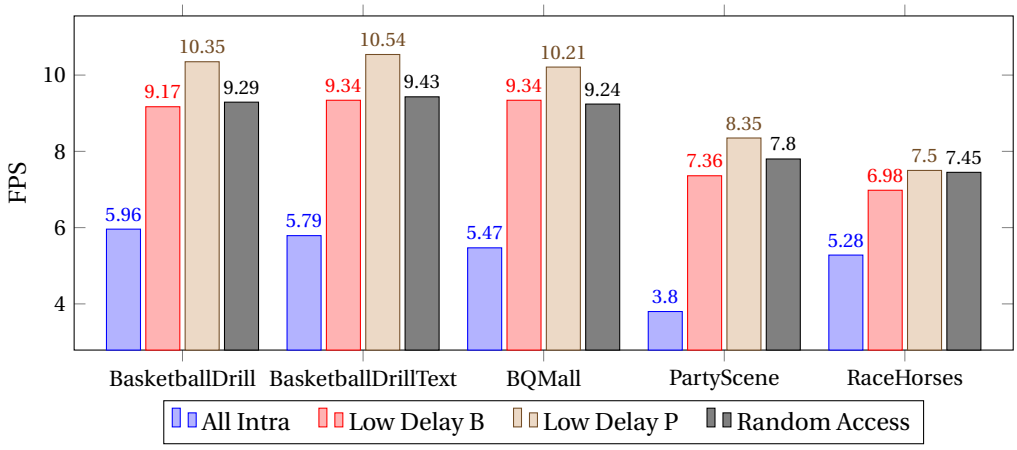


Figura 4.6: Velocidad de descodificación para secuencias de Clase C y QP 32.

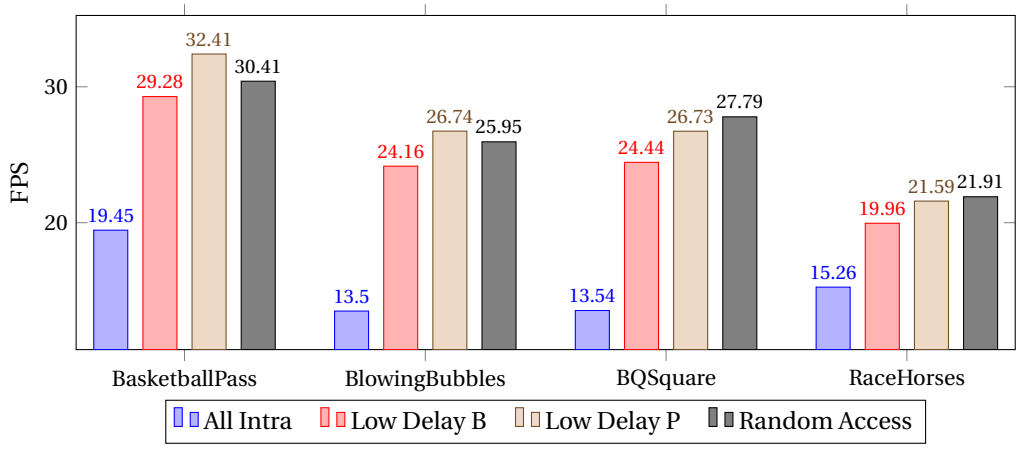


Figura 4.7: Velocidad de descodificación para secuencias de Clase D y QP 27.

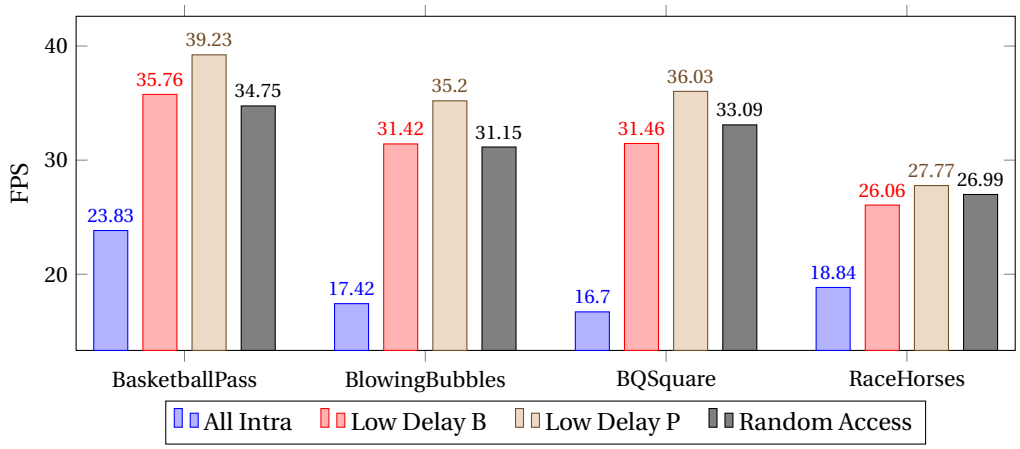


Figura 4.8: Velocidad de descodificación para secuencias de Clase D y QP 32.

4.5. Carga computacional

En este apartado se completa el estudio del rendimiento del descodificador openHEVC. En concreto, mediante el simulador, se analiza el porcentaje de carga computacional de cada bloque funcional del descodificador.

4.5.1. Automatización

En este caso el objetivo es obtener el número de ciclos de reloj que se emplean en cada función del código. De esta manera identificando que funciones pertenecen a cada bloque funcional se puede obtener un porcentaje de carga de cada bloque del descodificador.

Como indica la figura 4.9, la metodología es muy parecida a la expuesta en la sección 4.4. Pero ahora el objetivo es obtener un fichero (profile.csv) con los resultados del *profiler*.

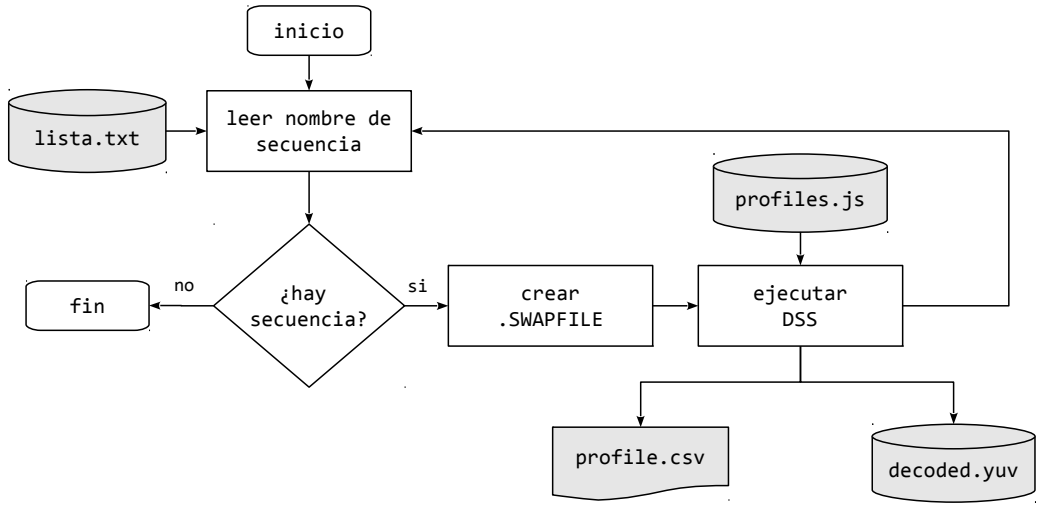


Figura 4.9: Esquema del proceso de obtención de los *profiles*.

El programa Python es idéntico al del emulador pero ahora el *script* de DSS (profiles.js) es algo más complejo. Su función es iniciar una sesión del *profiler*, para acto seguido ejecutar el programa openHEVC en el simulador. Después espera a la finalización del programa y guarda en un fichero CSV los resultados del la herramienta *profiler*.

Tras obtener todos los *profiles* de las secuencias ya se tienen datos suficientes para analizar el porcentaje de uso de cada bloque funcional. A continuación se describen los programas que se han escrito para realizar este análisis de forma automática.

En el listado 4.10 se muestra el contenido de openHEVC.func. Se trata de un archivo en formato INI[12] donde se describe la relación entre los bloques funcionales del descodificador (ver tabla 4.6) y las funciones C del código de openHEVC.

Listado 4.10: Fichero openHEVC.func.

```
# Block-Function description file for openHEVC.
#
# Each section (except DEFAULT) describes a decoder functional block.
# Functional blocks are described by functions.
# Each function must start with +(plus), -(minus) or .(dot)
#
# Meaning:
#   + Add including cycles
#   - Substract including cycles
#   . Add excluding cycles
#
# Includig cyles: Is the total number of cycles for all executions
#                 of the function including function calls.
# Excluding cyles: Is the total number of cycles for all executions
#                  of the function excluding function calls.
#
# Jesús Pablo Caño Velasco <jp.cano@alumnos.upm.es>
#

[DEFAULT]
# General information about the decoder.
# main = main(int, char * *)
main = hevc_decode_frame(struct AVCodecContext *, void *, int *, struct AVPacket
    *)


[ED]
```

[12] Los archivo INI son simples archivos de texto con una estructura básica compuesta por secciones y propiedades.

```
# Entropy decoder
+ff_hevc_significant_coeff_flag_decode(struct HEVCContext *, int, int, int, int,
    int)
+ff_hevc_coeff_abs_level_greater1_flag_decode(struct HEVCContext *, int, int, int,
     int, int)
+ff_hevc_last_significant_coeff_x_prefix_decode(struct HEVCContext *, int, int)
+ff_hevc_last_significant_coeff_y_prefix_decode(struct HEVCContext *, int, int)
+ff_hevc_coeff_sign_flag(struct HEVCContext *, unsigned char)
+ff_hevc_cbf_luma_decode(struct HEVCContext *, int)
+ff_hevc_coeff_abs_level_remaining(struct HEVCContext *, int, int)
+ff_hevc_prev_intra_luma_pred_flag_decode(struct HEVCContext *)
+ff_hevc_cbf_cb_cr_decode(struct HEVCContext *, int)
+ff_hevc_rem_intra_luma_pred_mode_decode(struct HEVCContext *)
+ff_hevc_transform_skip_flag_decode(struct HEVCContext *, int)
+ff_hevc_intra_chroma_pred_mode_decode(struct HEVCContext *)
+ff_hevc_part_mode_decode(struct HEVCContext *, int)
+ff_hevc_mpm_idx_decode(struct HEVCContext *)
+ff_hevc_coeff_abs_level_greater2_flag_decode(struct HEVCContext *, int, int, int)
+ff_hevc_split_transform_flag_decode(struct HEVCContext *, int)
+ff_hevc_split_coding_unit_flag_decode(struct HEVCContext *, int, int, int)
+ff_hevc_significant_coeff_group_flag_decode(struct HEVCContext *, int, int, int,
    int)
+ff_hevc_sao_offset_abs_decode(struct HEVCContext *)
+ff_hevc_last_significant_coeff_suffix_decode(struct HEVCContext *, int)
+ff_hevc_sao_type_idx_decode(struct HEVCContext *)
+ff_hevc_sao_merge_flag_decode(struct HEVCContext *)
+ff_hevc_sao_band_position_decode(struct HEVCContext *)
+ff_hevc_cabac_init(struct HEVCContext *)
+ff_hevc_sao_eo_class_decode(struct HEVCContext *)
+ff_hevc_end_of_slice_flag_decode(struct HEVCContext *)
+ff_hevc_sao_offset_sign_decode(struct HEVCContext *)

+ff_hevc_cu_transquant_bypass_flag_decode(HEVCContext *s)
+ff_hevc_skip_flag_decode(HEVCContext *s, int x_cb, int y_cb)
+ff_hevc_pred_mode_decode(HEVCContext *s)
+ff_hevc_pcm_flag_decode(HEVCContext *s)
+ff_hevc_no_residual_syntax_flag_decode(HEVCContext *s)

# In inter prediction profiles
+ff_hevc_merge_idx_decode(HEVCContext *s)
+ff_hevc_merge_flag_decode(HEVCContext *s)
+ff_hevc_inter_pred_idc_decode(HEVCContext *s, int nPbW, int nPbH)
+ff_hevc_ref_idx_lx_decode(HEVCContext *s, int num_ref_idx_lx)
+ff_hevc_mvp_lx_flag_decode(HEVCContext *s)


[ITQ]
# Inverse transform and quantization
+hls_residual_coding(struct HEVCContext *, int, int, int, enum ScanType, int)
-ff_hevc_transform_skip_flag_decode(struct HEVCContext *, int)
-ff_hevc_last_significant_coeff_x_prefix_decode(struct HEVCContext *, int, int)
-ff_hevc_last_significant_coeff_y_prefix_decode(struct HEVCContext *, int, int)
-ff_hevc_last_significant_coeff_suffix_decode(struct HEVCContext *, int)
-ff_hevc_significant_coeff_group_flag_decode(struct HEVCContext *, int, int, int,
    int)
-ff_hevc_significant_coeff_flag_decode(struct HEVCContext *, int, int, int, int,
    int)
-ff_hevc_coeff_abs_level_greater1_flag_decode(struct HEVCContext *, int, int, int,
     int, int)
-ff_hevc_coeff_abs_level_greater2_flag_decode(struct HEVCContext *, int, int, int)
-ff_hevc_coeff_sign_flag(struct HEVCContext *, unsigned char)
-ff_hevc_coeff_abs_level_remaining(struct HEVCContext *, int, int)


[IP]
# Intra prediction
+intra_pred_8_c(struct HEVCContext *, int, int, int, int)
+intra_prediction_unit(struct HEVCContext *, int, int, int)
-ff_hevc_prev_intra_luma_pred_flag_decode(struct HEVCContext *)
-ff_hevc_mpm_idx_decode(struct HEVCContext *)
```

```
-ff_hevc_rem_intra_luma_pred_mode_decode(struct HEVCContext *)
-ff_hevc_intra_chroma_pred_mode_decode(struct HEVCContext *)


[EP]
# Inter prediction
+hls_prediction_unit(struct HEVCContext *, int, int, int, int, int, int)
-ff_hevc_merge_idx_decode(HEVCContext *s)
-ff_hevc_merge_flag_decode(HEVCContext *s)
-ff_hevc_inter_pred_idc_decode(HEVCContext *s, int nPbW, int nPbH)
-ff_hevc_ref_idx_lx_decode(HEVCContext *s, int num_ref_idx_lx)
-ff_hevc_mvp_lx_flag_decode(HEVCContext *s)


[DF]
# Deblocking filter
+deblocking_boundary_strengths(struct HEVCContext *, int, int, int)
+deblocking_filter(struct HEVCContext *)


[SAO]
# Sampling Adaptative Offset filter
+hls_sao_param(struct HEVCContext *, int, int)
-ff_hevc_sao_merge_flag_decode(struct HEVCContext *)
-ff_hevc_sao_type_idx_decode(struct HEVCContext *)
-ff_hevc_sao_offset_abs_decode(struct HEVCContext *)
-ff_hevc_sao_offset_sign_decode(struct HEVCContext *)
-ff_hevc_sao_band_position_decode(struct HEVCContext *)
-ff_hevc_sao_eo_class_decode(struct HEVCContext *)
+sao_filter(struct HEVCContext *)
+av_picture_copy(struct AVPicture *, struct AVPicture *, enum AVPixelFormat, int,
    int)
```

Por otra parte, en el listado 4.11 se muestra la clase `Decana`, cuyo constructor recibe como argumento la ruta de un fichero .func como el mostrado en el listado 4.10. El método `getPercentages` recibe como parámetro el nombre de un fichero .csv y devuelve una lista con los porcentajes de carga de cada uno de los bloques funcionales.

El programa mostrado en el listado 4.12 hace uso de la clase `Decana` y tabula, en un documento Latex[13], los porcentajes de carga de los bloques funcionales del descodificador para cada una de las secuencias analizadas. En el apéndice B se muestran dichas tablas.

Tabla 4.6: Bloques funcionales del descodificador openHEVC.

Bloque funcional	Abreviatura
Entropía	ED
Transformada inversa y cuantificación	ITQ
Predicción intra	IP
Predicción inter	EP
Filtro antibloques	DF
Filtro SAO	SAO
Otros	OT

[13] Latex es un sistema de composición de textos compuesto por comandos en texto plano.

Listado 4.11: Clase Decana del programa de análisis de *profiles*.

```
import ConfigParser
import csv

class Decana (object):
    def __init__(self, funcfile):
        self.config = ConfigParser.ConfigParser(allow_no_value = True)
        self.config.optionxform = str # upcase sensitive
        self.config.read(funcfile)
        self.main = self.config.get('DEFAULT', 'main')

    def getFuncCycles(self, csvfile, function, token):
        '''
        Returns the number of cycles of 'function' in 'csvfile'.
        'token': + add inlcuding, - substract including, . add exlcuding.
        FIXME: implement a binary search.
        FIXME: Deal with profiles generated without Debug Scripting Server.
        '''
        with open(csvfile, 'rb') as f:
            reader = csv.reader(f, delimiter = ',')
            for row in reader:
                if row[1] == function:
                    if token == '+':
                        return float(row[10])
                    elif token == '-':
                        return -float(row[10])
                    elif token == '.':
                        return float(row[6])
            return 0

    def formatPercentages(self, floatvalue):
        '''
        Returns 'floatvalue' rounded up to 1 decimal place
        '''
        return float("{0:.1f}".format(floatvalue))

    def getPercentages(self, csvfile):
        '''
        Returns a list with the percentages of each decoder block according
        to 'funcfile'. The last element is the Others block percentage.
        '''
        result = []
        maincycles = self.getFuncCycles(csvfile, self.main, '+')
        for section in self.config.sections():
            cycles = 0
            for func in dict(self.config.items(section)):
                add = self.getFuncCycles(csvfile, func[1:], func[0])
                cycles += add
            result.append(self.formatPercentages(100 * cycles / maincycles))
        # Others block
        result.append(self.formatPercentages(100 - sum(result)))
        return result
```

Listado 4.12: Programa decana-blocks.py para analizar *profiles*.

```
from decana import Decana
import itertools

def main():
    configs = ['All Intra (AI\\_MAIN)',
               'Low Delay B (LD\\_MAIN)',
               'Low Delay P (LP\\_MAIN)',
               'Random Access (RA\\_MAIN)']

    paths = ['profiles/i_main/',
             'profiles/ld_main/',
             'profiles/lp_main/',
             'profiles/ra_main/']

```

```
    decana = Decana("openhevc.func.txt")

    tex = open('template.tex', 'r').read()

    tex = tex.replace('%%% 416x240 table %%%',
                      createtable(decana,'list.txt','416x240',
                                  configs, paths))
    tex = tex.replace('%%% 832x480 table %%%',
                      createtable(decana,'list.txt','832x480',
                                  configs, paths))
    tex = tex.replace('%%% func file %%%', open('openhevc.func.txt', 'r').read())

    with open("openHEVCfunctionalBlocks.tex", "w") as f:
        f.write(tex)

def secname(sec):
    '''Return the name of a sequence'''
    return sec.split('_')[0]
def secsize(sec):
    '''Return the size of a sequence'''
    return sec.split('_')[1]
def secqp(sec):
    '''Return the QP of a sequence'''
    return int(sec.split('_')[3][2:4])
def csvname(sequence):
    return 'TOTAL_' + sequence[0:-4] + '.csv'

def getSequences(listfile, size, qp):
    sequences = []
    with open(listfile, "r") as f:
        for line in f:
            line = line.strip()
            if line == '' or line[0] == '#': continue
            if size and secsize(line) != size: continue
            if qp and secqp(line) != qp: continue
            sequences.append({'name': secname(line),
                              'qp': secqp(line),
                              'csv': csvname(line)})
    return sequences

def createtable(decana, listfile, size, configs, paths):
    table = """
    \\begin{tabularx}{\\textwidth}{llXccccccc} \\\\
    \\toprule
    \\bf{Config} & \\bf{Sequence} & \\bf{QP} & \\bf{ED} & \\bf{ITQ}
    & \\bf{IP} & \\bf{EP} & \\bf{DF} & \\bf{SAO} & \\bf{OT} \\\\
    \\toprule
    """
    sequences = getSequences(listfile, size, None)
    for conf, path in itertools.izip(configs, paths):
        total = [0] * 7
        for sec in sequences:
            data = decana.getPercentages(path + sec['csv'])
            total = [sum(a) for a in zip(*[total,data])]
            table += '%s & %s & %d' % (conf, sec['name'], sec['qp'])
            for val in data: table += ' & ' + str(val)
            table += ' \\\\ \\hline '
        total = map(lambda x: float("{0:.1f}".format(x / len(sequences))), total)
        table += ' & & '
        for val in total: table += ' & \\bf{' + str(val) + '}'
        table += ' \\\\ \\\\ \hline '
    table +="\\end{tabularx}"
    return table

if __name__ == "__main__":
    main()
```

Listado 4.13: Plantilla Latex usada por el programa decana-blocks.py.

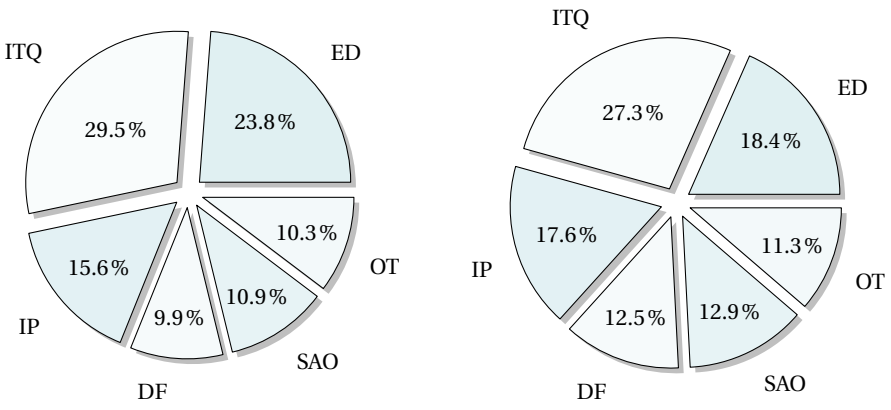
```
\documentclass[a4paper]{article}
\usepackage{fourier}
\usepackage[T1]{fontenc}
\usepackage{datetime}
\usepackage{tabularx}
\usepackage[labelfont=bf,font=small]{caption}
\pagestyle{fancy}
\usepackage{fancyhdr}
\lhead{\textsc{openHEVC functional blocks}}
\chead{}
\rhead{Generated \today, \currenttime}

\begin{document}
\section{Class D sequences (240p)}
\begin{table}[H]
\caption{Class D sequences (240p), percentage of computational load of each
    functional block}
%%% 416x240 table %%%
\end{table}
\section{Class C sequences (480p)}
\begin{table}[H]
\caption{Class C sequences (480p), percentage of computational load of each
    functional block}
%%% 832x480 table %%%
\end{table}
\section{Block-Function description file}
\begin{verbatim}
%%% func file %%%
\end{verbatim}
\end{document}
```

4.5.2. Resultados

En este apartado se muestran los resultados obtenidos con el proceso descrito en la sección 4.5.1. Las medidas se han realizado tomando las 10 primeras imágenes de cada secuencia. Sólo se muestran los resultados de las secuencias de clase C puesto que se ha comprobado que los porcentajes de las secuencias de clase D son similares.

En la figura 4.10 el uso de la predicción inter (EP) es nulo puesto que se trata de secuencias *All Intra*. La transformada inversa es el bloque que más consume (casi un 30 %), seguido de la entropía (20 %). El resto de carga es repartido casi equitativamente en los demás bloques.



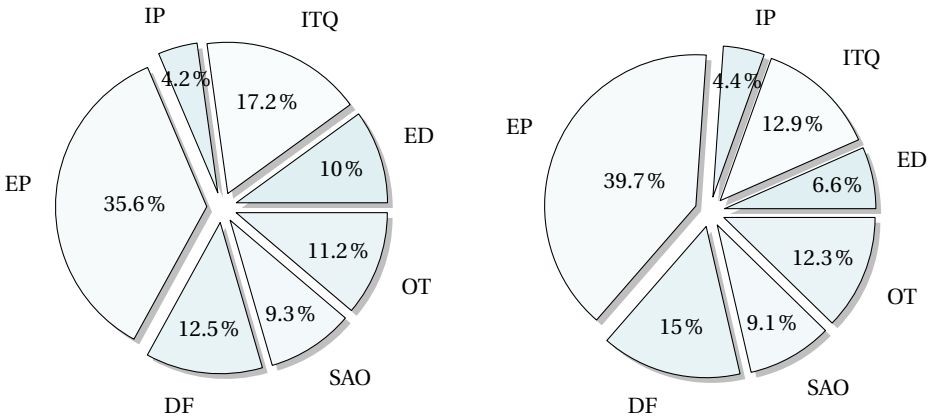
(a) QP 27 **(b)** QP 32

Figura 4.10: Porcentaje de carga computacional en secuencias Clase C *All Intra*.

En las secuencias *Low Delay B* (figura 4.11), *Low Delay P* (figura 4.12) y *Random Access* (figura

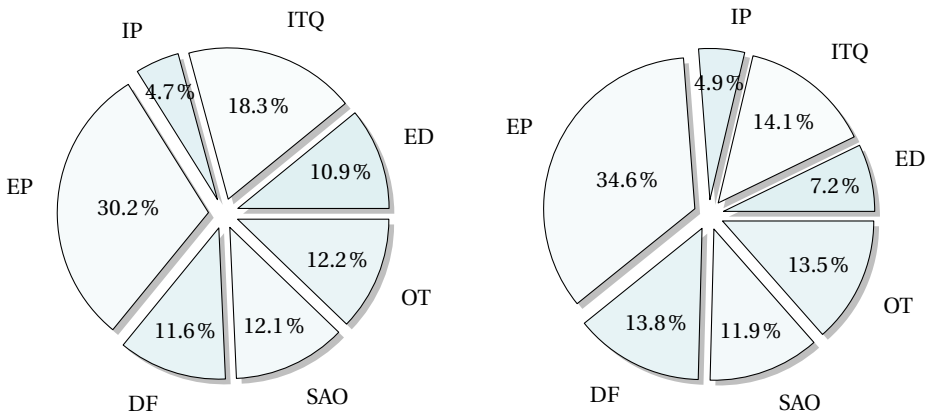
4.13), al tener predicción temporal, el bloque que más tiempo consume es la predicción inter (EP) (entre un 30 % y un 40 %). Este bloque es dependiente del QP, siendo un 4 % mayor para secuencias con QP 32.

Cabe mencionar que el filtro antibloques (DF) y el filtro SAO (SAO) se mantienen aproximadamente constantes para cualquier configuración (alrededor del 12 %). Aunque, por lo general, el DF tiene mas carga que SAO.



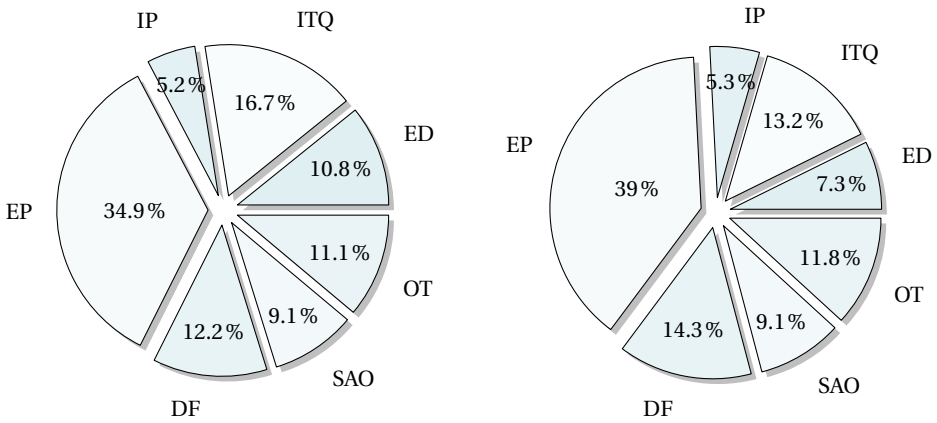
(a) QP 27 **(b)** QP 32

Figura 4.11: Porcentaje la carga computacional secuencias Clase C *Low Delay B*.



(a) QP 27 **(b)** QP 32

Figura 4.12: Porcentaje de carga computacional en secuencias Clase C *Low Delay P*.



(a) QP 27 **(b)** QP 32

Figura 4.13: Porcentaje de carga computacional en secuencias Clase C *Random Access*.

4.6. Comparación con el software de referencia HM 9.0

En esta sección se comparan los resultados obtenidos en este trabajo con los que se han obtenido en una implementación [14] del software de referencia (HM [8]) sobre la misma plataforma de desarrollo que se ha utilizado en este proyecto (DM6437EVM). En la tabla 4.7 se muestran las principales diferencias entre ambas implementaciones.

Tabla 4.7: Diferencias en el banco de pruebas de HM y openHEVC.

Características	HM	openHEVC
Lenguaje	C++	C
Imágenes (para FPS)	10	100
Imágenes (para porcentajes)	10	10
Sistema operativo	SYS/BIOS	DSP/BIOS
Versión de HEVC	HM 9.0	HM 10.0

Como se puede ver en la tabla 4.8, con openHEVC se consigue una ratio de mejora en la velocidad de descodificación de 2.8 para secuencias *All Intra* y 2.6 para secuencias *Low Delay P* y *Random Access*. Se observa que esta ratio es dependiente del QP siendo en las secuencias con QP 32 superior a las que tienen QP 27.

Pese a ello, en ninguna de las dos implementaciones se logra descodificar en tiempo real. Para ello, sería necesario llevar a cabo optimizaciones como las descritas en los trabajos [15, 16, 18].

Las diferencias en la velocidad de descodificación se pueden justificar por el hecho de que el objetivo de openHEVC es la optimización del rendimiento en la descodificación. En cambio, el objetivo de HM es la corrección, completitud y legibilidad del código.

Además el software de referencia HM está escrito en C++ mientras que openHEVC lo está en C. Según [27], el uso de algunas características de C++ como *polimorfismo*, *clases virtuales*, *excepciones*, *RTTI* y *casts dinámicos* pueden afectar negativamente al rendimiento en la ejecución del código, sobre todo si se usan incorrectamente. Por ello el compilador C++ de Texas Instruments es menos eficiente que el de C.

En la tabla 4.9 se muestra una comparación de la carga computacional de cada bloque funcional del descodificador en ambas implementaciones. En openHEVC se observa una menor

Tabla 4.8: Comparación de los FPS en HM y openHEVC (OH).

Secuencia	QP	All Intra			Low Delay P			Random Access		
		HM	OH	*ratio*	HM	OH	*ratio*	HM	OH	*ratio*
BasketballDrill	27	1.6	4.7	2.9	3.1	8.3	2.7	3.0	7.9	2.6
BasketballDrill	32	1.9	6.0	3.1	3.6	10.4	2.9	3.4	9.3	2.7
BQMall	27	1.6	4.5	2.8	3.5	8.3	2.4	3.2	8.0	2.5
BQMall	32	1.7	5.5	3.2	4.0	10.2	2.6	3.6	9.2	2.6
PartyScene	27	1.3	3.1	2.4	2.5	6.2	2.5	2.5	6.4	2.5
PartyScene	32	1.4	3.8	2.7	3.0	8.4	2.8	2.9	7.8	2.7
RaceHorses	27	1.7	4.5	2.6	2.3	5.8	2.5	2.3	6.0	2.6
RaceHorses	32	1.8	5.3	2.9	2.8	7.5	2.7	2.7	7.5	2.8

carga en la predicción *intra* (IP) en cualquier configuración. Por último, la carga de la transformada inversa (ITQ) es mayor, sobre todo en la configuración *All Intra*.

Tabla 4.9: Comparación del porcentaje de carga computacional en los bloques funcionales de HM y openHEVC (OH).

Bloque	All Intra		Low Delay P		Random Access	
	HM	OH	HM	OH	HM	OH
ED	18	21	9	16	9	15
ITQ	17	28	12	16	12	15
IP	32	17	11	5	13	5
EP	0	0	26	32	27	37
DF	14	11	15	13	14	13
SAO	4	12	5	12	5	9
OT	14	11	14	13	14	11

Capítulo 5

Conclusiones

En este capítulo se presentan las conclusiones de este proyecto fin de carrera. En la sección 5.1 se realiza una exposición de los resultados obtenidos en el capítulo previo. En la sección 5.2 se indican algunas de las posibles mejoras que se pueden realizar en trabajos futuros.

5.1. Exposición

En este proyecto fin de carrera se ha logrado el objetivo inicial de caracterizar el rendimiento de un descodificador HEVC. Se ha medido la velocidad de descodificación y la carga computacional de los bloques funcionales del descodificador para un amplio conjunto de secuencias. Además de migrar el descodificador openHEVC al DSP, se ha establecido una forma de realizar las medidas de rendimiento de forma automatizada. Esto último ha permitido reducir enormemente el tiempo necesario para lograr el objetivo.

Se ha mostrado que, en la plataforma utilizada, el DSP es capaz de ejecutar openHEVC y descodificar secuencias de video HEVC a una velocidad muy inferior al tiempo real y diferente en cada secuencia, dependiendo del tamaño de la imagen, el tipo de secuencia y la complejidad de ésta.

Se ha comprobado que el descodificador openHEVC puede descodificar secuencias a una velocidad aproximadamente 2.3 veces la de un descodificador basado en el código de referencia HM9.0 implementado en un trabajo anterior y en la misma plataforma.

Finalmente, con los resultados obtenidos en este Proyecto Fin de Carrera, el grupo GDEM ha escrito un artículo [20] que está pendiente de aceptación en *International Conference on Consumer Electronics 2014*.

5.2. Trabajos futuros

En esta sección se exploran algunas de las limitaciones de este trabajo fin de carrera y se presentan como posibles mejoras en trabajos futuros.

En el momento de escribir esta memoria, openHEVC es un proyecto en activo desarrollo. Por ello sería recomendable analizar el rendimiento de la última versión disponible, la cual incluye mejoras funcionales y de rendimiento. Gracias a la herramienta de control de versiones Git y a los programas de automatización de medidas, realizados en este proyecto, el tiempo necesario para el análisis del rendimiento del nuevo descodificador se reduce considerablemente.

En [19] se recopilan algunas técnicas para optimizar el rendimiento de descodificadores de vídeo en las plataformas con DSP. Se pueden aplicar estas técnicas a openHEVC y explotar con ello las capacidades del *hardware* del DSP.

Del mismo modo se puede optimizar la ejecución del descodificador utilizando las nuevas plataformas hardware que incluyen un DSP multinúcleo (como es el caso del DSP TMDSEVM6678L de Texas Instruments). De esta manera, se puede aprovechar la arquitectura multihilo de openHEVC y las capacidades de paralelización de HEVC (WPP y *tiles*).

Por último, puesto que la finalización del estándar HEVC está llegando a su término, el equipo JCT-VC ha propuesto las extensiones de vídeo escalable para HEVC con el nombre de

SHVC (*Scalable High-efficency Video Coding*) [9]. Actualmente sólo existe la implementación desarrollada durante el proceso de propuesta[1]. Se puede continuar la línea que se inició con el descodificador openSVC [1] e implementar un descodificador escalable para HEVC. Y finalmente, con el conocimiento adquirido con [17] y [15] se podría optimizar para plataformas con DSP.

[1] Disponible en https://hevc.hhi.fraunhofer.de/svn/svn_SHVCSoftware/.

Apéndice A

Tablas de FPS

En este apéndice se muestra la velocidad de descodificación de cada secuencia probada en el descodificador openHEVC. Estas tablas se han obtenido de forma automática con el programa descrito en la sección 4.5.1.

A.1. Secuencias de clase C

Tabla A.1: Ciclos CPU (x10^6) y FPS promedios en secuencias de clase C.

Secuencia	QP	All Intra		Low Delay B		Low Delay P		Rand. Access	
		FPS	CLKs	FPS	CLKs	FPS	CLKs	FPS	CLKs
BQMall	27	4.53	131	7,63	78	8.28	72	7.95	75
BQMall	32	5.47	109	9.34	64	10.21	58	9.24	64
BasketballDrillText	27	4.64	128	7.66	78	8.53	70	8.04	74
BasketballDrillText	32	5.79	103	9.34	64	10.54	56	9.43	63
BasketballDrill	27	4.67	127	7.48	79	8.33	71	7.86	76
BasketballDrill	32	5.96	100	9.17	65	10.35	57	9.29	64
PartyScene	27	3.06	194	5.67	105	6.17	96	6.37	93
PartyScene	32	3.80	157	7.36	81	8.35	71	7.80	76
RaceHorses	27	4.45	133	5.36	111	5.80	102	5.95	100
RaceHorses	32	5.28	112	6.98	85	7.50	79	7.45	80

A.2. Secuencias de clase D

Tabla A.2: Ciclos CPU (x10^6) y FPS promedios en secuencias de clase D.

Secuencia	QP	All Intra		Low Delay B		Low Delay P		Rand. Access	
		FPS	CLKs	FPS	CLKs	FPS	CLKs	FPS	CLKs
BQSquare	27	13.54	44	24.44	24	26.73	22	27.79	21
BQSquare	32	16.70	36	31.46	19	36.03	16	33.09	18
BasketballPass	27	19.45	31	29.28	20	32.41	18	30.41	20
BasketballPass	32	23.83	25	35.76	17	39.23	15	34.75	17
RaceHorses	27	15.26	39	19.96	30	21.59	28	21.91	27
RaceHorses	32	18.84	32	26.06	23	27.77	21	26.99	22
BlowingBubbles	27	13.50	44	24.16	25	26.74	22	25.95	23
BlowingBubbles	32	17.42	34	31.42	19	35.20	17	31.15	19

Apéndice B

Tablas de porcentajes

En este apéndice se muestran las tablas del porcentaje de carga de los bloques funcionales del descodificador openHEVC. Estas tablas se han obtenido de forma automática con el programa descrito en la sección 4.5.1.

B.1. Secuencias de clase C

Tabla B.1: Porcentaje de carga computacional en secuencias Clase C *All Intra*.

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQMall	22.6	29.1	16.3	0.0	10.4	10.9	10.7
BasketballDrillText	19.5	27.1	17.6	0.0	11.7	12.8	11.3
BasketballDrill	19.2	26.7	18.0	0.0	12.0	12.7	11.4
PartyScene	31.2	32.6	13.2	0.0	6.7	7.3	9.0
RaceHorses	26.5	32.0	13.0	0.0	8.8	10.7	9.0
Promedio	**23.8**	**29.5**	**15.6**	**0.0**	**9.9**	**10.9**	**10.3**

(a) QP 27

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQMall	18.0	26.9	18.1	0.0	12.8	12.4	11.8
BasketballDrillText	14.3	24.9	19.0	0.0	14.4	15.4	12.0
BasketballDrill	13.8	24.9	19.3	0.0	14.8	15.2	12.0
PartyScene	25.5	30.1	15.7	0.0	9.0	9.1	10.6
RaceHorses	20.2	29.5	15.7	0.0	11.3	12.6	10.7
Promedio	**18.4**	**27.3**	**17.6**	**0.0**	**12.5**	**12.9**	**11.3**

(b) QP 32

Tabla B.2: Porcentaje de carga de cada bloque funcional en secuencias Clase C *Low Delay B*.

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQMall	9.1	14.0	4.4	36.1	14.4	9.2	12.8
BasketballDrillText	6.9	16.1	4.3	36.3	13.8	11.1	11.5
BasketballDrill	6.9	17.0	4.4	37.2	13.8	9.2	11.5
PartyScene	15.4	19.7	4.0	33.9	9.7	6.7	10.6
RaceHorses	11.5	19.0	3.7	34.5	10.9	10.2	10.2
Promedio	**10.0**	**17.2**	**4.2**	**35.6**	**12.5**	**9.3**	**11.2**

(a) QP 27

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQMall	6.2	10.5	4.7	38.8	16.6	9.4	13.8
BasketballDrillText	4.6	12.6	4.1	40.4	16.3	9.4	12.6
BasketballDrill	4.4	12.7	4.2	41.2	16.1	9.0	12.4
PartyScene	10.3	14.7	4.6	38.5	12.4	7.8	11.7
RaceHorses	7.4	14.1	4.2	39.6	13.7	9.9	11.1
Promedio	**6.6**	**12.9**	**4.4**	**39.7**	**15.0**	**9.1**	**12.3**

(b) QP 32

Tabla B.3: Porcentaje de carga computacional en secuencias Clase C *Low Delay P*.

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQMall	9.7	14.8	4.7	32.1	12.7	12.3	13.7
BasketballDrillText	7.7	17.5	4.9	31.1	12.9	13.3	12.6
BasketballDrill	7.6	18.0	4.9	30.7	12.9	13.4	12.5
PartyScene	17.0	21.3	4.5	28.3	9.0	8.4	11.5
RaceHorses	12.6	20.1	4.3	28.7	10.5	13.0	10.8
Promedio	**10.9**	**18.3**	**4.7**	**30.2**	**11.6**	**12.1**	**12.2**

(a) QP 27

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQMall	6.7	11.4	5.0	35.4	14.5	12.2	14.8
BasketballDrillText	5.1	13.9	4.8	35.7	15.2	11.1	14.2
BasketballDrill	4.9	14.3	4.8	35.3	15.1	11.9	13.7
PartyScene	11.5	16.1	5.2	33.2	11.4	9.7	12.9
RaceHorses	7.9	14.9	4.7	33.4	12.9	14.5	11.7
Promedio	**7.2**	**14.1**	**4.9**	**34.6**	**13.8**	**11.9**	**13.5**

(b) QP 32

Tabla B.4: Porcentaje carga computacional en secuencias Clase C *Random Access*.

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQMall	9.5	14.1	4.7	37.6	13.3	9.1	11.7
BasketballDrillText	7.7	15.6	5.1	36.8	13.5	9.8	11.5
BasketballDrill	7.6	16.0	5.3	36.8	13.5	9.4	11.4
PartyScene	16.7	20.0	5.1	31.5	9.4	6.8	10.5
RaceHorses	12.6	17.9	5.6	32.0	11.2	10.5	10.2
Promedio	**10.8**	**16.7**	**5.2**	**34.9**	**12.2**	**9.1**	**11.1**

(a) QP 27

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQMall	6.5	11.3	4.8	40.9	15.2	8.9	12.4
BasketballDrillText	5.0	12.2	4.8	40.3	15.5	10.2	12.0
BasketballDrill	4.9	12.7	5.0	40.7	15.5	9.2	12.0
PartyScene	11.6	15.6	5.7	36.2	11.9	7.5	11.5
RaceHorses	8.4	14.2	6.0	37.0	13.6	9.8	11.0
Promedio	**7.3**	**13.2**	**5.3**	**39.0**	**14.3**	**9.1**	**11.8**

(b) QP 32

B.2. Secuencias de clase D

Tabla B.5: Porcentaje de carga computacional en secuencias Clase D *All Intra*.

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQSquare	29.7	29.1	15.0	0.0	7.9	8.0	10.3
BasketballPass	22.5	30.5	15.3	0.0	9.8	11.3	10.6
RaceHorses	27.6	31.9	13.3	0.0	7.9	9.8	9.5
BlowingBubbles	26.9	32.0	14.2	0.0	7.9	9.0	10.0
Promedio	**26.7**	**30.9**	**14.4**	**0.0**	**8.4**	**9.5**	**10.1**

(a) QP 27

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQSquare	24.9	26.0	17.6	0.0	10.2	9.4	11.9
BasketballPass	16.5	28.0	17.6	0.0	12.6	13.3	12.0
RaceHorses	21.0	29.5	16.0	0.0	10.7	11.5	11.3
BlowingBubbles	20.6	30.0	16.7	0.0	10.8	10.5	11.4
Promedio	**20.8**	**28.4**	**17.0**	**0.0**	**11.1**	**11.2**	**11.5**

(b) QP 32

Tabla B.6: Porcentaje carga computacional en secuencias Clase D *Low Delay B*.

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQSquare	11.8	12.8	3.1	41.7	10.1	9.3	11.2
BasketballPass	9.1	13.4	3.7	38.4	13.7	8.1	13.6
RaceHorses	11.8	17.4	3.5	37.4	9.8	9.0	11.1
BlowingBubbles	11.0	15.9	3.1	41.3	10.4	6.8	11.5
Promedio	**10.9**	**14.9**	**3.4**	**39.7**	**11.0**	**8.3**	**11.8**

(a) QP 27

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQSquare	7.9	8.6	3.7	46.4	12.5	8.7	12.2
BasketballPass	5.7	9.5	4.1	40.9	16.3	8.9	14.6
RaceHorses	7.8	12.9	4.0	40.8	12.3	10.3	11.9
BlowingBubbles	7.2	12.0	3.6	44.1	13.2	7.1	12.8
Promedio	**7.2**	**10.8**	**3.9**	**43.0**	**13.6**	**8.8**	**12.7**

(b) QP 32

Tabla B.7: Porcentaje de carga computacional en secuencias Clase D *Low Delay P*.

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQSquare	13.3	14.2	3.3	35.1	9.1	12.1	12.9
BasketballPass	9.7	14.1	4.1	35.4	12.0	10.3	14.4
RaceHorses	12.7	18.2	3.9	32.6	9.5	11.3	11.8
BlowingBubbles	12.0	17.0	3.5	35.2	9.5	10.5	12.3
Promedio	**11.9**	**15.9**	**3.7**	**34.6**	**10.0**	**11.1**	**12.8**

(a) QP 27

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQSquare	9.0	9.8	4.0	38.4	11.1	13.8	13.9
BasketballPass	6.1	10.3	4.3	39.0	14.2	10.6	15.5
RaceHorses	8.3	13.5	4.4	36.2	11.6	13.5	12.5
BlowingBubbles	8.0	13.0	4.1	39.2	12.0	10.0	13.7
Promedio	**7.8**	**11.7**	**4.2**	**38.2**	**12.2**	**12.0**	**13.9**

(b) QP 32

Tabla B.8: Porcentaje de carga computacional en secuencias Clase D *Random Access*.

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQSquare	12.7	13.5	3.4	40.3	9.8	9.3	11.0
BasketballPass	9.3	12.7	4.4	39.8	13.1	8.4	12.3
RaceHorses	12.9	17.1	5.2	35.4	9.9	8.5	11.0
BlowingBubbles	11.9	16.1	3.8	40.6	9.9	6.9	10.8
Promedio	**11.7**	**14.8**	**4.2**	**39.0**	**10.7**	**8.3**	**11.3**

(a) QP 27

Secuencia	ED	ITQ	IP	EP	DF	SAO	OT
BQSquare	8.7	9.5	3.9	45.2	12.1	8.5	12.1
BasketballPass	6.0	9.6	4.4	43.6	15.0	8.3	13.1
RaceHorses	8.8	13.4	5.7	39.5	12.3	8.6	11.7
BlowingBubbles	7.7	12.2	4.0	44.6	12.3	7.6	11.6
Promedio	**7.8**	**11.2**	**4.5**	**43.2**	**12.9**	**8.2**	**12.2**

(b) QP 32

Bibliografía y referencias

[1] Código fuente en C que implementa un descodificador compatible con el anexo G de la norma H.264. http://sourceforge.net/projects/opensvcdecoder.

[2] Libav, biblioteca multiplataforma para convertir y manipular una gran variedad de formatos y protocolos multimedia. http://libav.org.

[3] openHEVC, descodificador HEVC open source programado en C para fines de investigación. http://github.com/OpenHEVC.

[4] Frank Bossen. Common test conditions and software reference configurations. http://phenix.int-evry.fr/jct/doc_end_user/current_document.php?id=5890, 2012.

[5] Bossen, F. and Bross, B. and Suhring, K. and Flynn, D. HEVC Complexity and Implementation Analysis. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1685–1696, 2012. Disponible en http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6317152.

[6] Jesús P. Caño. openHEVC DSP, migración de openHEVC al procesador digital de señal TMS320DM6437. http://github.com/jpcano/openHEVC.

[7] Chih-Ming Fu and Alshina, E. and Alshin, A. and Yu-Wen Huang and Ching-Yeh Chen and Chia-Yang Tsai and Chih-Wei Hsu and Shaw-Min Lei and Jeong-Hoon Park and Woo-Jin Han. Sample Adaptive Offset in the HEVC Standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1755–1764, 2012. Disponible en http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?tp=&arnumber=6324411.

[8] Fraunhofer Institute. HM 9.0. Software de referencia HEVC. http://hevc.hhi.fraunhofer.de/.

[9] ISO/IEC. Joint Call for Proposals on Scalable Video Coding Extensions of High Efficiency Video Coding (HEVC). http://mpeg.chiariglione.org/standards/mpeg-h/high-efficiency-video-coding/joint-call-proposals-scalable-video-coding-extensions, Julio 2012.

[10] ITU-T. Recommendation ITU-T H.265. http://www.itu.int/rec/T-REC-H.265, April 2013.

[11] G. Martres, M. Raulet, S. Tomperi, F. Pescador Del Oso, and Jean Le Feuvre. Open tools for packaging, streaming and playback of HEVC content. Submitted to ACM Multimedia 2013 Open Source Sofware Competition, 2013.

[12] Guillaume Martres. Google summer of code 2012, un descodificador HEVC para Libav. http://www.googlemelange.com/gsoc/project/google/gsoc2012/smarter/8001.

[13] Ohm, J. and Sullivan, G.J. High efficiency video coding: the next frontier in video compression [Standards in a Nutshell]. *Signal Processing Magazine, IEEE*, 30(1):152–158, 2013. Disponible en http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6375943.

[14] F. Pescador, M. Chavarrías, M.J. Garrido, E. Juárez, and C. Sanz. Complexity analysis of an HEVC de coder based on a Digital Signal Processor. Aceptada en Transaction on Consumer Electronics, 2013.

[15] F. Pescador, E. Juarez, M. Raulet, and C. Sanz. A DSP based H.264/SVC decoder for a multimedia terminal. *IEEE Transactions on Consumer Electronics*, 57(2):705–712, May 2011. También disponible en http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5722651.

[16] F. Pescador, G. Maturana, M.J. Garrido, E. Juarez, and C. Sanz. An H.264 video decoder based on a DM6437 DSP. *IEEE Transactions on Consumer Electronics*, 55(1):205–212, February 2009. También disponible en http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5012351.

[17] F. Pescador, David Samper, Matías J. Garrido, Eduardo Juarez, and Mederic Blestel. A DSP based SVC IP STB using Open SVC Decoder. Braunschweig, Alemania, June 2010. 14th IEEE International Symposium on Consumer Electronics ISCE 2010. También disponible en http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5523708.

[18] F. Pescador, C. Sanz, M.J. Garrido, C. Santos, and R. Antoniello. A DSP based IP set-top box for home entertainment. *IEEE Transactions on Consumer Electronics*, 52(1):254–262, February 2006. También disponible en http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1605055.

[19] Fernando Pescador del Oso. *Contribución a las metodologías de optimización del tiempo de ejecución de algoritmos de descodificación de video sobre DSPs*. PhD thesis, Universidad Politécnica de Madrid. Escuela Universitaria de Ingeniería Técnica de Telecomunicación, 2011. También disponible en http://oa.upm.es/8712/.

[20] Pescador, F. and Caño, J.P. and Garrido, M.J. and Juarez, E. and Raulet, M. A DSP HEVC Decoder Implementation Based on openHEVC. Pendiente de aceptación en International Conference on Consumer Electronics 2014.

[21] Pourazad, M.T. and Doutre, C. and Azimi, M. and Nasiopoulos, P. HEVC: The New Gold Standard for Video Compression: How Does HEVC Compare with H.264/AVC? *Consumer Electronics Magazine, IEEE*, 1(3):36–46, 2012. Disponible en http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6222536.

[22] Spectrum Digital. DM6437EVM Support Page. Disponible en http://c6000.spectrumdigital.com/evmdm6437.

[23] Spectrum Digital. *TMS320DM6437 Evaluation Module Technical Reference (Revision C)*, December 2006. Disponible en http://c6000.spectrumdigital.com/evmdm6437/reve/files/EVMDM6437_TechRef.pdf.

[24] Sullivan, G.J. and Ohm, J. and Woo-Jin Han and Wiegand, T. Overview of the High Efficiency Video Coding (HEVC) Standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668, 2012. Disponible en http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6316136.

[25] Texas Instruments. C99 in TI compilers. http://processors.wiki.ti.com/index.php?title=C99_in_TI_Compilers&oldid=148313.

[26] Texas Instruments. Code Composer Studio, entorno de desarrollo integrado. http://www.ti.com/tool/ccstudio.

[27] Texas Instruments. Overview of C++ support in TI compilers. http://processors.wiki.ti.com/index.php?title=Overview_of_C%2B%2B_Support_in_TI_Compilers&oldid=129206.

[28] Texas Instruments. *TMS320DM6437 Digital Media Processor*, June 2008. Datasheet SPRS345D disponible en http://www.ti.com/litv/pdf/sprs345d.

[29] Texas Instruments. *TMS320C64x+ DSP Cache User's Guide (Revision B)*, February 2009. Disponible en http://www.ti.com/litv/pdf/spru862b.

[30] Texas Instruments. *TMS320C6000 Optimization Workshop. Discussion Notes*, March 2011. Disponible en http://software-dl.ti.com/trainingTTO/trainingTTO_public_sw/op6000/op6000_v1.51/op6000_student_guide_v1.51.pdf.

[31] Texas Instruments. *TMS320C6000 Programmer's Guide (Revision K)*, July 2011. Disponible en http://www.ti.com/litv/pdf/spru198k.

[32] Texas Instruments. *TMS320C6000 Optimizing Compiler v7.4 User's Guide (Revision U)*, July 2012. Disponible en http://www.ti.com/litv/pdf/spru187u.

[33] Vanne, J. and Viitanen, M. and Hamalainen, T.D. and Hallapuro, A. Comparative Rate-Distortion-Complexity Analysis of HEVC and AVC Video Codecs. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1885–1898, 2012. Disponible en http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6324420.

[34] Viitanen, M. and Vanne, J. and Hamalainen, T.D. and Gabbouj, M. and Lainema, J. Complexity analysis of next-generation HEVC decoder. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 882–885, 2012. Disponible en http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6272182.

Acrónimos

ALU	*Arithmetic Logic Unit*
API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
ASIC	*Application Specific Integrated Circuit*
AVC	*Advanced Video Coding*
CABAC	*Context Adaptative Binary Arithmetic Coding*
CAN	*Controller Area Network*
CAVLC	*Context Adaptative Variable Length Coding*
CB	*Coding Block*
CCS	*Code Composer Studio*
CELL	*Cell Broadband Engine Architecture*
CPU	*Central Processing Unit*
CTB	*Coding Tree Block*
CTS	*Clear To Send*
CTU	*Coding Tree Unit*
CU	*Coding Unit*
DAC	*Digital Analog Converter*
DCT	*Discrete Cosine Transform*
DDR	*Double Data Rate memory*
DF	*Deblocking Filter*
DMA	*Direct Memory Access*
DSP	*Digital Signal Processor*
DSS	*Debug Scripting Server*
DWARF	*Debugging With Attributed Record Formats*
ED	*Entropy Decoder*
EDMA	*Enhanced Direct Memory Access*
EMAC	*Ethernet Medium Access Controller*
EMIF	*Exernal Memory InterFace*
EP	*intEr Prediction*
FLC	*Fixed Length Code*
FMO	*Flexible Macroblock Ordering*
FPGA	*Field Programmable Gate Array*
FPS	*Frames Per Second*

GDEM *Grupo de Diseño Electrónico y Microelectrónico*

GPIO *General Purpose I/O*

GPP *General Purpose Processor*

GSoC *Google Summer of Code*

HECC *High-End Can Controller*

HEVC *High Efficency Video Coding*

INI *Windows INItialization file*

HM *HEVC test Model*

I^2C *Inter-Integrated Circuit*

IDMA *Internal Direct Memory Access*

IEC *International Electrotechnical Commission*

IP *Intra Prediction*

ISO *International Organization for Standardization*

ITQ *Inverse Transform and Quantization*

ITU *International Telecommunication Union*

JCT-VC *Joint Collaborative Team on Video Coding*

JTAG *Joint Test Action Group*

McASP *Multichannel Audio Serial Port*

McBSP *Multichannel Buffered Serial Port*

MD5 *Message-Digest Algorithm 5*

MIPS *Microprocessor without Interlocked Pipeline Stages*

MPEG *Moving Picture Experts Group*

NAL *Network Abstraction Layer*

NAND *Negated AND memory*

OSD *On Screen Display*

OT *OThers block*

PC *Personal Computer*

PCB *Printed Circuit Board*

PCI *Peripheral Component Interconnect*

PDF *Portable Document Format*

PLL *Phase-Locked Loop*

POSIX *Portable Operating System Interface*

PU *Prediction Unit*

PWM *Pulse Width Modulator*

QP *Quantisation Parameter*

RGB *Red, Green and Blue*

RISC *Reduced Instruction Set Computer*

RTS	*Request To Send*
RTTI	*Run-Time Type Information*
S/PDIF	*Sony/Philips Digital Interface Format*
SAO	*Sampling Adaptive Offset filter*
SHVC	*Scalable High-efficency Video Coding*
SRAM	*Static Random Access Memory*
TTY	*TeleTYpewriter*
TU	*Transform Unit*
UART	*Universal Asynchronous Receiver-Transmitter*
UDTV	*Ultra High Definition Television*
UHD	*Ultra High Definition*
VCEG	*Video Coding Experts Group*
VCL	*Video Coding Layer*
VCL	*Variable Lenght Code*
VLIW	*Very Long Instruction Word*
VPSS	*Video Processing SubSystem*
WPP	*Wavefront Parallel Processing*